MODELING AND PREDICTING SOFTWARE BEHAVIORS

A Dissertation
Presented to
The Academic Faculty

By

James F. Bowring

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology

December 2006

MODELING AND PREDICTING SOFTWARE BEHAVIORS

Approved by:

Dr. Mary Jean Harrold, Advisor
College of Computing
Georgia Institute of Technology

Dr. James M. Rehg, Advisor
College of Computing
Georgia Institute of Technology

Dr. Spencer Rugaber
College of Computing
Georgia Institute of Technology

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Dr. H. Andy Podgurski
Computer Science Department
Case Western Reserve University

Date Approved: August 10, 2006

1.6180339887

I dedicate this work to Sharla Aylene Gould: Thank you, my Love!

ACKNOWLEDGEMENTS

In 1996, at the age of 46, I announced to my wife, Sharla Gould, that I would sell my interest in my general-contracting business and return to school to seek an undergraduate degree and a graduate degree in Architecture. Ten years later, I have earned a Ph.D. and, more importantly, I have learned to appreciate the courage of Sharla in her support and encouragement of my journey. During these ten years, our daughter Dylann attended high school and then college. Sharla, a teacher by profession, worked all day with middle-school students and then had both husband and daughter as students at home. With my entry into graduate school in 2001, we bought a second house in Atlanta where I would live two weeks at a time between visits home. Dylann moved to Athens to attend the University of Georgia, and suddenly Sharla had an "empty nest." Five years later including over 100,000 miles of driving and innumerable telephone calls and emails, I am returning home for good, Dylann has moved to San Diego, and my heart is full of appreciation for Sharla's sacrifices and love.

Mary Jean Harrold, my advisor, invited me into her research group when Georgia Tech admitted me to the Ph.D. program. In the year before I arrived, she assisted me in my successful application for a graduate fellowship. She has been an indefatigable mentor and supporter without whom I may not have succeeded. She is the first and only person I have met who could match and exceed my own output. Mary Jean was always available for consultation or for editing or for just talking and for this support above-and-beyond the call of duty, I am in her debt. I hold her in the highest esteem and I intend to emulate her approach to academic pursuits.

My brother, Dr. Samuel Bowring of MIT, has cheered me on and provided me immeasurable support at each stage of the process. His insights from his experiences as a student and as a seasoned advisor helped to keep me grounded and relaxed when I questioned my own progress.

Jim Rehg, my co-advisor, was at first taken aback by my interest in applying concepts and techniques from automated speech recognition and machine learning to software engineering. However, once I successfully communicated my ideas, he became a valued critic and supporter. His advice on the details of my research was invaluable, but he also provided me with cogent insights into how to plan an academic career.

Spencer Rugaber has been critically supportive of me during my tenure at Georgia Tech. I took or audited most of the courses he teaches and, as a result, I am very much prepared to teach the same material myself. He has also provided many insights into my work. In particular, he spent many hours reading my dissertation draft with red pen in hand and I am grateful for his careful attention to detail and his many helpful suggestions.

When I embarked on my Ph.D. studies, Alex Orso was a research scientist working with Mary Jean Harrold and helping her to manage several research projects. Although we butted heads from the outset, Alex was instrumental in guiding the research that led to my first paper, co-authored by him and Mary Jean Harrold. He is now an assistant professor at Georgia Tech. As a member of my reading committee, he has asked the hard questions that drove me to deeper insights and that helped me to improve how I communicate my ideas. He also provided many invaluable suggestions for improving the dissertation.

My first introduction to Andy Podgurski was as an author of several papers I found intriguing because of their focus on aggregate analysis of program execution data using techniques from statistical machine learning. Unbeknownst to him, these papers provided me with the inspiration to follow my own predilections. We did meet eventually at conferences, and he graciously agreed to serve as the outside member of my oral defense committee. His work in cluster analysis applied to program executions is foundational to my own. Andy provided many useful and detailed suggestions for the text of my dissertation.

Jim Jones is a fellow Ph.D. student who began the year before I did and helped me to acclimate to the graduate program and to Mary Jean Harrold's Aristotle Research Group. We progressed through the various Ph.D. milestones nearly in tandem and are now collaborating on a research project that we both include in our dissertations. Jim has a unique ability to provide unvarnished insights and he has done so selflessly for me. We have become good friends and I anticipate that we will also become collaborating colleagues.

Taweesup Apiwattanapong and I began our studies with Mary Jean Harrold at the same time. Term, as he calls himself, had just spent some weeks in a Thai monastery and arrived shorn of hair and I had just spend a year away from school consulting as a forensic expert in building construction. An odd couple, to say the least, but we have supported each other as we achieve the requisite milestones. In the summer of 2003, we met everyday for an hour or two to read and dissect the material for our qualifier exam. Term always provided good questions and insights, and helped me to advance my ideas.

Term also rents a room from me and we have shared four years of observing each other's cultural quirks and cooking.

Donglin Liang was the senior Ph.D. student when I arrived. He is now an assistant professor at the University of Minnesota. Donglin was an inspirational example of scientific inquisitiveness—he asked good questions and he was open to new ideas. I learned also from his experiences as he proposed and defended.

In general, all the students and faculty at Georgia Tech with whom I have interacted have been friendly and supportive. This congenial atmosphere has been refreshing and supportive for me.

My Ph.D. studies were funded in full and I want to acknowledge these invaluable sources. I had four principal benefactors. I received a three-year National Defense Science and Engineering Graduate (NDSEG) Fellowship. I received a one-year Phi Kappa Phi graduate fellowship. I received a Georgia Tech President's Fellowship for four years. Finally, Georgia Tech and Mary Jean Harrold's research grants provided me with two years as a research assistant. I am deeply grateful for this support.

Finally, as I leave Georgia Tech and assume my new duties at the College of Charleston, I want to thank the faculty of the Department of Computer Science for hiring me as an assistant professor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Software systems will eventually contribute to their own maintenance using implementations of self-awareness. Understanding how to specify, model, and implement software with a sense of self is a daunting problem. This research draws inspiration from the automatic functioning of a gimbal---a self-righting mechanical device that supports an object and maintains the orientation of this object with respect to gravity independently of its immediate operating environment. A software gimbal exhibits a self-righting feature that provisions software with two auxiliary mechanisms: a historical mechanism and a reflective mechanism. The historical mechanism consists of behavior classifiers trained on statistical models of data that are collected from executions of the program that exhibit known behaviors of the program. The reflective mechanism uses the historical mechanism to assess an ongoing or selected execution.

This dissertation presents techniques for the identification and modeling of program execution features as statistical models. It further demonstrates how statistical machine-learning techniques can be used to manipulate these models and to construct behavior classifiers that can automatically detect and label known program behaviors and detect new unknown behaviors. The thesis is that statistical summaries of data collected from a software program's executions can model and predict external behaviors of the program.

This dissertation presents three control-flow features and one value-flow feature of program executions that can be modeled as stochastic processes exhibiting the Markov property. A technique for building automated behavior classifiers from these models is

detailed. Empirical studies demonstrating the efficacy of this approach are presented. The use of these techniques in example software engineering applications in the categories of software testing and failure detection are described.

1 INTRODUCTION

Thesis:

> Statistical summaries of data collected from a software program's executions can model and predict external behaviors of the program.

Software pervades our built environment.[1] The growth in the number of active CPUs and executing programs has outpaced the ability of humans to maintain and manage digital systems without automated assistance. Ideally, digital systems will eventually exhibit some degree of self-awareness and contribute to their own maintenance. There exist many mechanical systems that use digital systems for monitoring and maintenance as, for example, the on-board computers in most automobiles. However, these computers themselves are subject to anomalous behaviors as well. Understanding how to specify, model, and implement software with a sense of self is a daunting problem. One promising approach to this problem is to develop meta-strategies for self-awareness. For example, Murdoch and Goel have proposed techniques for the self-adaptation of specialized software agents using meta-case-based reasoning [71]. As another example, IBM's initiatives with autonomic computing address this idea of meta-strategies for the management of systems on a large scale [6, 53]. The IBM approach models the network of resources required by customers as a biological entity that responds and changes dynamically (i.e., autonomically) to the needs of the customers. Researchers also study domain-specific self-aware systems in robotics,

[1] The "built environment" is a commonly-used architectural term used to illuminate the human impact on the natural environment. See, for example, Christopher Alexander's *The Origins of Pattern Theory* [6].

control systems, software architecture, and fault-tolerant computing. These research approaches concentrate on the creation of high-level models to assess program behavior with regard to the operating environment and then provide for simple self-maintenance (e.g., [30, 31, 42]).

However, self-awareness begins at home, so to speak, and there is scant progress in providing for the self-awareness of individual programs such as those driving robots and other scientific devices or even those in a desktop environment. Researchers have been intrigued by self-awareness for a number of years. For example, Forrest and colleagues proposed a sense of self for Unix processes as an "artificial immune system [41]." The authors seek to design security systems that mimic salient features of natural immune systems such as probabilistic, on-line detection of viruses. In current practice, software agents can intercept outputs produced by the execution of terminating programs for data collection and analysis. These outputs include binary dumps of executions, memory traces, program state information to inform execution replays, and user reports. Software engineers analyze these outputs individually or as a group. However, programs cannot yet evaluate their own behavior in any detail either during or after execution. There is an emerging consensus that the development of self-awareness for software will enhance dependability and safety for a wide range of applications (e.g., [16, 40, 45]). Support for this viewpoint comes from research communities such as the *Workshop on Self-healing Systems (WOSS)* [3], the *Workshop on Design and Evolution of Autonomic Application Software (DEAS)* [43], and the *Self-Repairing and Self-Configurable Distributed Systems Conference (RCDS)* [1]. Additionally, there are substantial industrial initiatives from International Business Machines (IBM), Microsoft, Sun Microsystems,

and the Defense Advanced Research Projects Agency (DARPA), for example, that support this viewpoint.

The scope of the problems associated with reifying software self-awareness is vast and researchers are just beginning to define and solve the foundational problems. As a first step towards a practical solution for the vision of software self-awareness, I have undertaken in this research to examine in detail how to exploit correlations between the internal behaviors of a program and two external behaviors: "passing" and "failing." This dissertation presents my approach to implementing software self-awareness in this restricted context and presents my research wherein I have developed specific modeling techniques for summarizing program execution data and specific automated techniques for the construction of effective behavior classifiers using the resultant models as data elements.

1.1 Motivation

My vision of a practical solution to implementing software self-awareness draws inspiration from the automatic functioning of a *gimbal*—a mechanical device that supports an object and maintains the orientation of this object with respect to gravity independently of its immediate operating environment. A gimbal is self-righting. A gimbal for software will also provide for self-righting, but to do so, a software gimbal will at a minimum require knowledge of baseline behaviors.

My approach to designing a software gimbal that exhibits a self-righting feature is to provision software with two auxiliary mechanisms: a historical mechanism and a reflective mechanism. The historical mechanism consists of behavior classifiers trained

on statistical models of data that are collected from executions of the program exhibiting known behaviors. The reflective mechanism uses the historical mechanism to assess a selected execution. This scheme is an automated pattern-recognition system. The patterns consist of statistical models of program-execution data. I extract these models from data collected during the execution of a program. Then I use statistical machine-learning techniques to manipulate these models and to construct behavior classifiers that can automatically detect and predict specific learned behaviors.

This approach of using a historical and a reflective mechanism gives rise to two basic research questions:

- How can we efficiently extract statistical models from program executions and map them to known program behaviors?
- How can we enable software to evaluate its own behavior in terms of these models?

This dissertation addresses these two questions by proposing and evaluating specific answers. First, I introduce an example program that will serve to illustrate the techniques and results that are presented throughout this dissertation.

1.1.1 An Example Program

Weinberg first specified and Meyers later studied a simple program to illustrate the complexity of designing testing strategies [97]:

> The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral [68].

```
Procedure TriangleType
 S1  char *returnString = "not triangle\n";
 S2  if (((i + j)> k)&&((j + k)> i)&&((k + i)> j))
 S3     if ((i + j + k) == (3 * i))
 S4        returnString =  "equilateral\n";
        else
 S5        if ((i != j) && (j != k) && (k != i))
 S6           returnString = "scalene\n";
           else
 S7           returnString = "isosceles\n";
           endif
        endif
     endif
 S8  return returnString;
```

Figure 1.1. Example procedure TriangleType.

Figure 1.1 shows an implementation of this specification as the procedure *TriangleType*. This implementation extends Meyer's specification with the additional printed output of "not triangle" for inputs that do not form a triangle such as (1, 2, 3) or (0, 1, 1). Thus, this program encodes four distinct specified behaviors, as characterized by the four possible outputs: "not triangle," "equilateral," "scalene," and "isosceles."

1.1.2 Behavior modeling

As described above, the historical mechanism of a software gimbal consists of behavior classifiers trained on statistical models of data collected from executions of the program that exhibit known behaviors. The principle requirement is that these summaries model the underlying data in a useful way. The notion of usefulness is at the heart of modeling since a model must capture the essence of the data in some simplifying

way that also provides for the comparison and manipulation of these models as surrogates for the underlying data.

These requirements illustrate the usual trade-off between efficiency and completeness. The benefits of summaries are their reduced size and reduced associated computational costs. However, summaries are generalizations and thus contain less information than the summarized data. Thus, the cost of using summaries is a reduction in precision and the benefit of using summaries is a potential increase in efficiency.

In this discussion of software behaviors, the summaries at any point on this continuum are models or patterns extracted from the known data describing a program's executions. In the case of the example program *Triangle,* a reasonable goal for the historical mechanism of a software gimbal is to have four models: one for each characteristic internal behaviors of the program. Each model summarizes the essence of all executions that exhibit the identified internal behavior. Thus, for example, a model named *model-equilateral-decision* will summarize the execution of the program every time the input integers are equal and greater than zero. The characteristic internal behavior of *Triangle* when its inputs are three identical natural numbers is to transform the inputs into the output "equilateral." However, *model-equilateral-decision* will not explicitly include the inputs. Rather, the model will summarize this internal behavior from salient features of other data collected about the actual execution of *Triangle* while it acts to type a triangle as equilateral.

One of the core problems with this modeling approach is in determining which data features can best summarize and characterize behaviors and under what conditions. These data features might measure the interaction of *Triangle* with the computing

environment relative to system calls, resource usage, and environment variables, for example. Alternatively, these data features might measure the internal workings of the program relative to control-flow and data-flow. In this dissertation, I concentrate on modeling the data features that measure the internal workings of a program. I seek first the essence of external behaviors that will occur whenever and wherever the program executes. My intuition is that just as a ship's gimbal provides for the operational integrity of the supported compass regardless of the body of water or the time of day, so too might a software gimbal provide for the operational integrity of the software regardless of the specific operating environment.

In recent years, software engineers have begun to explore the uses of statistical models of software behavior extracted from a dynamic analysis of a program's execution. These works range from studies of event profiles and event spectra to invariant detection to actual reports of failure from users. One trend is the use of machine learning techniques is to analyze and model execution data for building automated classifiers of software behaviors. Representative publications include References [18, 20, 23, 32, 46, 48, 49, 70, 79]. I discuss and evaluate these works in Chapter 2.

1.2 Overview of the Dissertation

As stated above, the potential scope of solutions for software self-awareness is vast. In this dissertation, I examine in detail a part of one potential solution: how to correlate the internal behaviors of a program with two clearly-defined and observable external behaviors: "passing" and "failing." This dissertation presents techniques that first extract statistical models of a program's internal behaviors from data collected

during that program's execution and then map them to these two external behaviors. These techniques focus on four transitions in the control-flow and value-flow of a program and model features of the collected data as stochastic processes exhibiting the Markov property. These four features are branch profiles, branch-to-branch profiles, method-to-method profiles, and databin-transition profiles.

The research first examines how well a single Markov model of each of these features can represent the internal behaviors of a program by training and evaluating an automatic behavior classifier from an aggregation of executions known to be "passing" or "failing." In particular, this dissertation presents research describing how these statistical models can be manipulated using statistical machine-learning techniques to construct, refine, and deploy such automatic behavior classifiers. One important aspect of this approach is the use of an "active-learning" paradigm to improve the performance of the resultant classifiers.

The presented methodologies can be applied to software engineering problems, and this dissertation describes four such categories of use. First, in the category of software testing, an application of these techniques to automated test suite augmentation is detailed. Second, in the evaluation of deployed software, an application of these techniques to detecting anomalous behaviors is described. Third, in a novel approach to debugging, these techniques are combined with a fault-localization technique to perform parallel, in contrast to sequential, debugging. Finally, for software self-awareness, patterns in method-to-method calls were studied as predictors of external behaviors.

Finally, this dissertation presents a series of empirical studies that validate the thesis and support the example applications of these techniques in four categories of use.

1.3 Contributions

There are four major contributions of this work.

- **Development of effective behavior classifiers:** This work demonstrates that four features of program execution data can be modeled as stochastic processes that exhibit the Markov property—branch profiles, branch-to-branch profiles, method-to-method profiles, and databin-transition profiles. This work further develops techniques to construct automatically behavior classifiers from these models that effectively discriminate between "passing" and "failing" executions.
- **Prototype software gimbal:** This work produced a prototype software gimbal that supports modeling execution data and the construction and evaluation of classifiers for empirical studies of the properties of various features of program execution data.
- **Example applications:** This work shows that the techniques developed are useful in four categories of software engineering applications: software testing, failure detection, fault-localization, and software self-awareness.
- **Empirical studies:** This work presents two sets of empirical studies. The first set validates the thesis that statistical summaries of data collected from a software program's executions can model and predict external behaviors of the program. The second set supports the use of the developed techniques in the example applications.

2 ANALYSIS AND MODELING OF PROGRAM BEHAVIORS

The techniques presented in this dissertation extend work in the areas of static, dynamic, and aggregate program analysis through the leveraging of work in pattern recognition and machine learning. This section presents this background material and describes previous research on the automated classification of software behaviors.

2.1 The problem of program behaviors

The concept of program behavior is widely used in software engineering. Parnas defines behavior as one of several ways to describe programs: "Behavioral descriptions describe some aspects of the executions of a program; they generally do not describe how the program is constructed from component programs [75]." In particular, Parnas presents *before/after* descriptions of behavior that ignore the intermediate states of an execution. These descriptions map initial safe states of a program to a set of possible executions. Bartussek and Parnas have proposed behavioral verification techniques based on assertions about execution traces [12]. Kowal, a requirements engineer, provides a parallel description of verifiable behavior as "consist[ing] of one or more units of activity supported by stored data and input data (and/or control) flows that accomplish some visible or audible result, or perform an activity within a process(or) that can be verified by some type of inspection [57]."

Testing engineers also have working definitions of program behaviors [68]. Functional or black-box testing tests the external behaviors of a program by how it transforms certain inputs to outputs. This view is similar to Parnas' before/after

descriptions of behavior. Structural or glass-box testing seeks to verify these external behaviors of a program by inspecting internal static and internal dynamic properties (i.e., internal behaviors) of the program. These internal behaviors could be measured using execution traces, as Parnas suggests. For example, Parnas proposes the use of assertions about legal traces and about trace equivalency as verification techniques.

Thus, there are two basic ways to understand the term "program behavior." On the one hand, the *external behavior* of a program can be described by saying that it executes correctly or incorrectly, that it passes or fails its functional tests. On the other hand, the *internal behavior* of a program can be described by measuring the properties statement coverage, path coverage, and branch profiles. On the one hand, the internal behavior of a program affects the external behavior (e.g., "passing" or "failing,") and on the other hand, the external inputs to the program control, in part, the internal behavior of the program. This interplay makes software engineering difficult because while is possible to engineer a program to process a specific input correctly, it is hard to anticipate all potential inputs and therefore the resultant external behavior they might induce.

Thus, a fundamental problem of software engineering is the translation of the requirements and their specified external behaviors into a program that behaves internally as intended. Formally, this requires that this translation be an *onto function*, where each actual internal behavior of the program is the image of a requirement. However, in practice, the translation is generally not an onto function, and programs may exhibit a different set of internal behaviors from those specified. For example, during the testing phase, the program may correctly exhibit every specified internal behavior, yet after deployment, users report bugs and crashes. In this case, there exist additional internal

behaviors beyond those specified. One goal of testing programs is to inform the development process to refine both the domain (i.e., the requirements) and the range (i.e., the required internal behaviors) towards the point that this translation does become an onto function. To achieve this goal, developers can use the testing to refine iteratively both the requirements and the software program.

Another example of a technique that may reduce the cost of a development iteration is the use of requirements monitoring, where high-level monitors of events such as licensure provide feedback to the developers [39]. In this case, engineers install requirements monitors to collect and analyze information about a program's run-time environment, such as a licensing database, to detect flaws in the assumptions framing the requirements. The collected data can aid the developer in pinpointing process flaws that customers may or may not report. In turn, the developer can immediately focus on correcting these flaws for future releases.

To reduce debugging costs and to provide for the systematic measurement of behaviors across development iterations, Bates proposes the Event-Based Behavioral Abstraction (EBBA) that uses a specific language to specify high-level event patterns in an event stream produced by probes in a program [14]. EBBA provides a developer with a tool to specify patterns of events to recognize particular behaviors that may be subject to change during program evolution. Thus, the developer can reuse the behavior models in subsequent iterations, reducing the per-iteration costs.

Liang and Xu have proposed the specific monitoring of scenarios [4] during program execution via the specification of scenario-specific properties [62]. The authors describe a monitoring technique that allows developers to compare scenarios identified

during requirements analysis to observed execution behaviors. They propose the use of this technique to reduce the costs of debugging programs, and thus to reduce the costs of development iterations.

As a final example, Barnett and Schulte propose an executable specification for a component that runs in parallel with the monitored component [11]. A goal of this verification is to guarantee that a specific execution of the monitored component is a behavioral subtype of its specification. The term "behavioral subtype" is defined by Liskov and Wing to mean that a subtype preserves the supertype methods' behaviors and that the subtype enforces the supertype's constraints and invariants [64]. This form of run-time verification monitors the compliance of the component with the interface contracts without direct instrumentation so as to detect whether the component has behaved as a subtype of a specified behavior. Again, the automated collection of behavioral data informs the development process and may provide for reduced costs.

2.1.1 My approach to program behaviors

These examples illustrate the practical difficulties inherent in refining the translation of the requirements or specified behaviors into a program that behaves internally as intended. This dissertation demonstrates techniques that can build statistical models of the internal behaviors of a program from an individual execution or from aggregations of executions of the program. This dissertation then demonstrates that there can be a strong correlation between these models of internal behavior and two specific external behaviors: "passing" and "failing." *Passing* behavior denotes that the program executed with an input and produced the expected outcome. *Failing* behavior denotes the

opposite: the outcome was not as expected. This dissertation presents techniques that demonstrate that there can be a measurable and significant correlation between statistical models of internal behavior and these two external behaviors. This result is independent of any specific input or outcome—the internal behaviors of the executing program predict the external behavior as either "passing" or "failing." These techniques extend work in the areas of dynamic and aggregate program analysis. In the next sections, I give an overview of this work and existing approaches to program-behavior analysis.

2.2 Program analysis

To model the internal behaviors of an executing program requires the ability to collect data about the execution, which in turn requires program analysis techniques that allow us to reason about its static and dynamic properties. The state of the practice in software engineering provides suitable static and dynamic analysis techniques for programs. *Static analysis* is the analysis of software source or object code performed without executing the program. Static analysis techniques provide information about program structures and potential flows within those structures, for example. *Dynamic analysis* techniques are performed using data collected during a program's execution. Dynamic analysis techniques support, for example, the collection of execution data such as the frequency of statement executions and the subsequent analysis of these data.

This dissertation restricts the discourse to specific forms of computer programs. This class of programs is written in an imperative language, such as C, and designed to execute and to terminate on a single processing thread in a deterministic fashion. These

programs are written according to the dictates of modern structured analysis and coding [67].

2.2.1 Control-flow

The *control-flow* of a program is an abstraction of all possible sequences of program statements in a program's execution. In structured programming, procedures (methods or functions) can contain calls to other procedures that then return to the calling procedure at a site in the control flow immediately after the site of the originating call unless the call is interupted. Of course, in the meantime, the called procedure can call other procedures with the same call-and-return rule. By convention in the C-language, we name the top-level procedure "main," which directs the activity of the program.

Because the control-flow under consideration models a deterministic process, directed graphs are well suited for modeling the flow. Two graph structures represent control-flow in a program:

> **Definition 2.1:** A *control-flow graph* (*CFG*) $G = (V, E)$ is a directed graph of a procedure. Each node in the set of nodes, V, represents a program statement. Each edge in the set of directed edges, E, represents the potential flow of control between two statements. Thus, a directed edge between two nodes, u and v, is $u \rightarrow v$ and represents the potential flow of control from u to v. An edge may also be annotated with a label, as for example, to show the value of a predicate statement that induced control to flow along the edge. Set V contains two special nodes: *Entry*, with no

predecessor and from which every node is reachable, and *Exit*, with no successor and reachable from every node (after [5, 60, 77]).

Definition 2.2: An *interprocedural control flow graph* (*ICFG*) *G = (V, E, P)* is a control-flow graph composed of the *CFG*s of one or more procedures plus a set *P* of pairs of procedure call (*C*) and return (*R*) edges connecting the component *CFG*s. *P* contains pairs *(C, R)* of call and return edges. Edge *C* connects a call site in one procedure *M1* with the *Entry* node of the *CFG* of another procedure *M2*. Edge *R* connects the *Exit* node of procedure *M2* with the return site in procedure *M1* (after [50, 59, 66, 92]).

As graphical representations of program control-flow, we define the meaning of paths and subpaths in these graphs.

Definition 2.3: A *path* in a *CFG* is a sequence of one or more graph edges $(x_0, x_1), (x_1, x_2), ..., (x_{n-1}, x_n)$ where x_0 = the *Entry* node and x_n = the *Exit* node. In the path sequence, each terminal node of an edge is identical to the initial node in the next edge in the path. This path is denoted by $x_0, x_1, x_2, ..., x_{n-1}, x_n$ and has a path length of *n* (after [84, 86]). Note that it is not possible to determine in general whether a path in a CFG can be executed [98].

Definition 2.4: A *subpath* in a *CFG* is any continuous sub-sequence of graph edges within a *path*.

On the left side of Figure 2.1 is a listing of the example procedure *TriangleType* introduced in Figure 1.1, and on the right side of Figure 2.1 is the corresponding CFG constructed according to the definition above. Each node in the CFG is labeled with the statement number it represents. The arrows in the CFG represent the directed flow of control. The predicates *S3, S4* and *S5* each have a "true" (T) and a "false" (F) directed edge to successor nodes. Additionally, there is a back-edge from *S8* to *S3* denoting potential looping behavior. While not shown as a figure, an ICFG is composed of CFGs, as defined above.

```
Procedure TriangleType
 S1  char *returnString = "not triangle\n";
 S2  if (((i + j)> k)&&((j + k)> i)&&((k + i)> j))
 S3     if ((i + j + k) == (3 * i))
 S4        returnString =  "equilateral\n";
        else
 S5        if ((i != j) && (j != k) && (k != i))
 S6           returnString = "scalene\n";
           else
 S7           returnString = "isosceles\n";
           endif
        endif
     endif
 S8  return returnString;
```

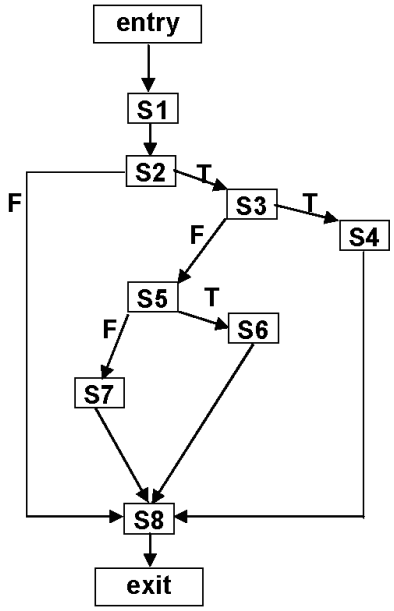


Figure 2.1. Example procedure TriangleType and its control-flow graph (CFG).

2.2.2 Data-flow and value-flow

Data-flow in a sequentially-executed program such as those considered in this dissertation is closely related to control-flow. The nodes of a CFG can be annotated with information about the definitions and uses of each variable. A variable *definition* occurs when a statement assigns a value to a variable, such as with an assignment statement or a read statement. A variable *use* occurs when a variable is used without being modified, as for example in a calculation or a predicate. Data-flow analysis seeks to understand how variables change during program execution.

As an example of the application of data-flow analysis, Laski and Korel present criteria for test selection derived from properties of data-flow that are based on definition-use chains in such a CFG [60]. A *definition-use chain* consists of a definition of a variable and all the uses of that variable reachable along subpaths in the CFG from that definition without any intervening re-definitions. The authors describe a white-box[2] testing strategy that activates parts of a data-flow model by exercising corresponding control-flow paths. The authors show that "data flow testing provides a finer test completeness measure than branch testing while avoiding the path explosion problem typical for path testing."

It is also possible to consider the flow of values of individual data objects or variables as a feature of program behaviors. Researchers have explored how such *value-flows* can be closely related to control-flows (e.g., [17, 34, 83, 100]). For example, Xie and Notkin present value spectra as a way to measure behavioral differences in program executions [100]. *Value spectra* measure properties of sets of variable values comprising

[2] White-box testing tests properties of an implementation [68].

the parameter list for user-function[3] calls and returns during program execution. These value spectra are used as a way to summarize the program's state. They demonstrate the use of value spectra in regression test selection by showing how value spectra can sometimes expose the root of a behavioral change.

Value-flows also inform the *Daikon* invariant detector,[4] which is a software tool that implements the dynamic detection of invariants in a program. An *invariant* is a program property that remains unchanged at points in a program. *Daikon* is used to execute a program and monitor values that variables take. Then, *Daikon* computes program properties that were invariant for the monitored executions [37]. This research uses *Daikon* to extract the databin transition feature in the empirical studies reported in Chapter 7.

2.2.3 Dynamic analysis and instrumentation

Static program analysis as described above computes program properties for all potential executions of the program without actually executing the program. Internal program behaviors are descriptions of potential program dynamics during execution. However, to measure actual program behaviors requires the use of dynamic program analysis. One use of static analysis is to provide the details of a program's structure that can be used in dynamic analysis to observe some properties of an execution. Just as engineers in other disciplines insert probes into their structures to detect the dynamic behavior of the structure under a load, a standard approach in software engineering is to insert measurement probes, or instrumentation [26], into the code of a program that will

[3] A user-function is a function explicitly defined in the source code in contrast to a library function.
[4] http://pag.csail.mit.edu/daikon/

report dynamic events. Dynamic analysis of programs exposes properties of executing programs by the inspection of internal events monitored by these probes [10].

The definitions of CFG and ICFG provide a way to reason about where in a program to insert these measurement probes for collecting data about the program's execution. The probes are themselves statements in the instrumented version of the program. Testing engineers design measurement probes to control and minimize side effects to the program to those associated with probe outputs and increased execution overhead. Because each probe incurs an expense in program execution overhead, the number and complexity of probes directly influences the costs of data collection. Researchers have developed techniques to minimize the number of probes by studying the structures of CFGs. For example, Knuth formalized algorithms for the efficient placement of probes to count either nodes (vertex profile) or edges (edge profile) in CFGs [55, 56]. Other reductions are possible. For example, the nodes of the CFG represent basic blocks[5] of statements, then one probe per basic block will provide coverage information for every statement in the basic block.

The most basic unit of measurable internal behavior is the output of a single probe. In general, the measurement of internal behaviors requires a set of probes. For example, consider the internal behavior that describes whether the statements $S = \{s_1, s_2, ..., s_n\}$ execute at least once. This internal behavior is called *statement coverage* of S. To verify this behavior, a probe inserted adjacent to each statement in S can log the statements' executions.

[5] A maximal sequence of instructions with a single entry point, single exit point, and no internal branches.

The logging function for each probe can be as simple as setting a bit in a binary array X, initially containing "0"s, mapped to S. Then, to verify that B occurred, we inspect X. Alternatively, the logging function could output an identifying token to an event stream E, creating a trace of the execution in terms of this logging function. In this case, verification of B involves the inspection of E.

2.2.4 Measuring program flows

In general, dynamic analysis depends upon the collection and summarization of data produced by measurement probes during program execution. These probes produce an event stream or flow that we can capture, sample, or process as needed. For example, to gather the frequency or profile of an event, we count the event's occurrence during program execution. As another example, it is possible to design probes that produce a trace from which we can replay the execution of the program [9]. Each program execution may exhibit differing event-flow sequences for a given set of probes due to different inputs and synergies between the program, its runtime environment, and the user. To facilitate measurement, we define models of flows that capture their essence relative to specified behavioral properties. Representative summaries of such program flows are branch profiles and method profiles [96], as well as relative frequencies of events (often called spectra) such as path spectra [85]. The use of models represents a trade-off between the expenses of preserving the event stream E on the one hand, and gaining the efficiency of summarization, on the other hand. A parallel trade-off occurs on the axis of information content. With gains in efficiency due to summarization comes

a loss of information—a summarization cannot inform a full reconstruction of an event stream.

2.2.5 Monitoring behaviors

Researchers have proposed that a potentially low-cost source for discovering the range of behaviors of a program is the history of all executions of the program after its deployment (e.g., [2, 18, 73, 76]). In one commonly proposed scenario, developers monitor data collected from executions via Internet connectivity. Monitoring deployed software and detecting behaviors is difficult for a number of reasons. Among these are technical problems of scale related to the identification of the data to collect, the instrumentation of programs, and the aggregation of the resultant data. An important computational problem is that the range or state space of internal behaviors for all executions of all copies of a given deployed program is potentially intractable. Furthermore, the precise determination of whether the internal behavior of *any single execution* exhibits a specific external behavior is in itself complex, even with full knowledge of the inputs and outcome. First, we must determine which requirement(s) the inputs to the specific execution are exercising. Given the complexity of many programs, this determination may itself be intractable. Second, once we know the specific external behavior to expect from these inputs, we must evaluate the outcome of the execution to confirm that the behavior occurred.

2.3 Modeling internal program behaviors

One promising approach to solving these problems is the application of statistical machine-learning techniques to model internal program behaviors from aggregate

execution data.[6] It is interesting to note that the survey paper entitled "Machine Learning and Software Engineering [101]" from 2002, focuses exclusively on automated software development processes (e.g., [13, 22, 25, 28, 36]). This survey does not reference the recent use of machine learning techniques by researchers exploring aggregate program behaviors. This work is explored in Section 2.3.2.

2.3.1 Pattern recognition and cluster analysis

Automated pattern classification and recognition depend on a process of information reduction that supports a classification decision to assign a label to a dataset. *Information reduction* is simply generalization as contrasted to specialization. A *pattern* is a generalization of properties of several specific data points that represents all of them as a class with a label. Consequently, once a classifier labels a specific dataset using a generalization or pattern, it is not possible to reconstruct the specific dataset from that label. In turn, the dataset itself is a summary or model of some feature of an object or event for which a classification is sought. There is a well-established design cycle for building automated pattern recognition systems as detailed in Reference [35]. The principal stages are (1) data collection, (2) feature choice, (3) model choice, (4) training, (5) evaluation, and (6) assessing computational complexity.

In this dissertation, the facts of deterministic programs and static and dynamic analysis methods together constrain the universe of discourse for the first two stages. For example, there is a known set of program features such as statements, predicates, and

[6] Aggregate execution data refers to data collected from a set of executions of the same program induced by identical or by differing inputs.

paths. We have formal analysis techniques, as discussed above, for reasoning about and measuring each of these features.

With respect to the third stage, the choice of models is both science and art, as model-builders often rely on their domain knowledge and experience as well as the mathematical properties of the data. For example, Cook and Wolf demonstrated the power of Markov chains in their work on software process discovery where they modeled software engineering processes using finite state machines [28]. In their work, they process event streams captured from human activity. The interpretation of their results is dependent on their domain knowledge of the software engineering process. Whittaker and Poore use Markov chains to model software usage from specifications prior to implementation [99]. The authors' domain knowledge of Cleanroom [81] requirements analysis and specification techniques informed the model selection. Baldi and colleagues go a step further and apply probabilistic methods to modeling the Internet and the Web, using Markov chains, Hidden Markov models, and Bayesian networks [8]. The authors argue that the dynamic and evolving interplay of users, databases, and software components on the Internet "yields uncertain and incomplete measurements and data," suggesting the use of these probabilistic models. Again, domain knowledge of the mechanics of the Internet informed the authors' model choice and their model structure.

The roots of automated pattern recognition or machine learning are in cluster analysis, which matured during the latter half of the twentieth century as a specialty in many research areas. According to Anderberg, who produced a survey and self-described distillation of the field in 1973, these areas included the life sciences, behavioral and social sciences, earth sciences, medicine, engineering (artificial intelligence and systems

science), and information and policy sciences [7]. Automated clustering algorithms were a natural outgrowth of cluster analysis as demonstrated in recent books such as Reference [35]. Of particular usefulness to the work described in this dissertation, as detailed in subsequent chapters, is the development of active learning as an improvement over batch learning by Cohn and colleagues [27]. In *batch learning*, the classifier trains once on a fixed quantity, or batch, of manually-labeled training data and cannot adapt to evolving datasets. In *active learning*, the classifier trains incrementally on a series of labeled data elements. During each iteration of learning, the current classifier chooses from the pool of unlabeled data to predict those elements that would most significantly extend the range of behaviors that it can classify. The human supervisor then labels these selected elements and adds them to the training set for the next round of learning.

2.3.2 Applications to software behavior analysis

As mentioned in the Introduction, the application of machine learning techniques such as automated classification to software engineering problems is relatively new. Representative publications include References [18, 20, 23, 32, 46, 48, 49, 70, 79].

My work builds on this prior work in a number of ways. First, all of this prior work adopts a classical batch-learning approach. My approach uses both the batch- and active-learning paradigms. The advantage of active learning in the software-engineering context is the ability to make effective use of the limited resources that are available for analyzing and labeling program execution data. For example, consider the scenario wherein we train a classifier to recognize and label each of a set of program executions as "passing" or "failing," by training on data collected about the internal behavior of the

executions. We then ask the classifier to label a new program execution. The classifier reports that it has a very low confidence in its decision. With the active-learning paradigm, a human then provides the correct label and the classifier trains on the additional data. Alternatively, if the classifier has a high confidence in its label, it proceeds normally. Because of this preferential selection, the scope of the classifier will dynamically extend beyond the scope provided by a batch-learning paradigm.

A second way in which my work builds on this prior work is in focusing on the specific properties and contributions of individual features of the data collected from executing programs. In many cases, this prior work does not segregate individual features for study. Haran and colleagues do isolate method calls, as described in the next paragraph [48]. Dickinson and colleagues show that clustering of executions based on method call profiles or method-to-method[7] profiles can isolate failing executions [32, 79]. The authors do not provide for automated refinement of the classifiers nor do they investigate other features. Podgurski and colleagues extend this work with an approach using multivariate visualization to aid the developer in determining whether the failing executions in a given cluster have related causes [79]. However, the authors select from a spectrum of 500 features without isolating individual features.

There exist many alternative statistical learning methods to analyze program executions. These methods share a goal of developing useful characterizations of program internal behaviors both singly and in aggregate. Harder, Mellen, and Ernst automatically classify software behaviors using an operational differencing technique [49]. Their method extracts formal operational abstractions from statistical summaries of program executions and uses them to automate the augmentation of test suites. Brun and

[7] The authors use the term function caller/callee profiles.

Ernst use dynamic invariant detection to extract program properties relevant to revealing faults and then apply batch learning techniques to rank and select these properties [23]. The properties they select are themselves formed from a large number of disparate features and the authors' focus is only on fault-localization and not on program behavior in general. Sekar and colleagues propose a mechanism to implement model-carrying code [91] focused on high-level models of security-relevant behavior. They propose techniques to extract automatically models from system call histories external to the program itself by using finite state automata. Haran and colleagues use random forests of method calls to model and predict behaviors of one subject [48]. The authors examine a suite of features and then do investigate a specific feature, method-call profiles, but they do not evaluate this feature's robustness with an active learning environment.

Researchers in fault localization also use statistical properties of aggregated executions. Jones and colleagues evaluate a set of program executions that contain both failing and passing instances by counting for each program statement the number of program executions that exercised that statement at least once. Then, using the relative frequencies of these counts and the knowledge of whether each execution passed or failed, they identify those statements most likely to be the cause of the fault [54]. Liblit and colleagues use statistical sampling and relative frequencies to prioritize a list of program predicates as fault indicators [63]. In both cases, the techniques become less effective in characterizing behaviors as they succeed in isolating faults. This deterioration is a by-product of any prioritization scheme: the top-most item expresses the studied indicator, but another indicator suited to the properties of the remaining items

might better order them. For example, tree-based classifiers branch on a unique feature at each node [35].

This research focuses on four features of program execution data and studies them individually as stochastic processes that exhibit the Markov property. This research differs from the work described above because of this focus on individual features and because of a successful classifier training technique that leverages probabilistic properties of models of these features. One benefit of studying features individually is to learn their costs and predictive powers independently of other features. Then, ensembles of individual features can be constructed intentionally to leverage their joint predictive properties. The features studied here are represented with real values, and consequently there is an inherent mathematical notion of a metric to measure similarity both absolutely and probabilistically. This contrasts with features that have nominal values, such as the external requirements of a program. For these features, there is no obvious way to compare similarity and to classify sets of these features requires parsing by, for example, a decision tree [35]. The probabilistic approach afforded by real-valued features enables dynamic classifier refinement in the presence of new program executions.

3 AUTOMATED CLASSIFIER CONSTRUCTION

As stated in the Introduction,[8] the first of two research questions motivating this research is: How can we efficiently extract statistical models from program executions and map them to known program behaviors? This mapping of data to behavior labels is a classification process. Classification identifies an execution as belonging to a specific class or category. In this context, a behavior classifier is a map from an execution feature, such as branch profiles, to a label for program behavior, such as "passing" or "failing." Three tasks must be addressed during classifier design: (1) identifying a set of features, (2) selecting a classifier architecture, and (3) implementing a learning technique for training the classifier using labeled data [69]. In Chapter 4, I describe the specific class of features explored in this research. However, to provide a motivating context, I first describe the architecture and automated training scheme of a classifier for external program behaviors derived from generic features of data collected from an executing program.

3.1 Approach

The approach used in this research is to leverage the techniques and tools developed for automated pattern recognition and classification. The automated training of a behavior classifier requires a set of training instances and their class labels. For the purposes of this dissertation, there are two class labels corresponding to two external behaviors of programs—"passing" and "failing." The general procedure for training a classifier is straightforward. First, the procedure selects a subject program and a set of

[8] See section 1.1: Motivation.

inputs to the program—each input will induce an execution of the program for which the external behavior label is known. Second, the procedure instruments the program so that each execution produces its own data according to some specification, such as branch profiling. Third, the instrumented program is executed with one of the selected inputs; the procedure collects and summarizes the resultant probe data and provides the label denoting the known external behavior of the execution. This collection of labeled data is the raw material from which the procedure will extract patterns that characterize, or map to, the external behaviors of the program. Finally, the procedure trains a classifier using this labeled data. A goal is that a classifier trained successfully with these data will be able to label correctly the data produced by the program when it executes on a new input.

3.2 Automatic construction of classifiers for a program

This research has produced a two-stage technique that builds automatically a classifier for a program's external behaviors. Figure 3.1 shows an overview of the technique as a dataflow diagram.

> **Definition 3.1:** A *dataflow diagram* represents the processes, data stores, and external entities in a system and shows the connecting flows of data. Processes or transformations of data are represented by circles, data stores by pairs of parallel horizontal lines, and external entities by rectangles. Arrows represent one or more data items described by a label on the arrow (after [80, 93].)

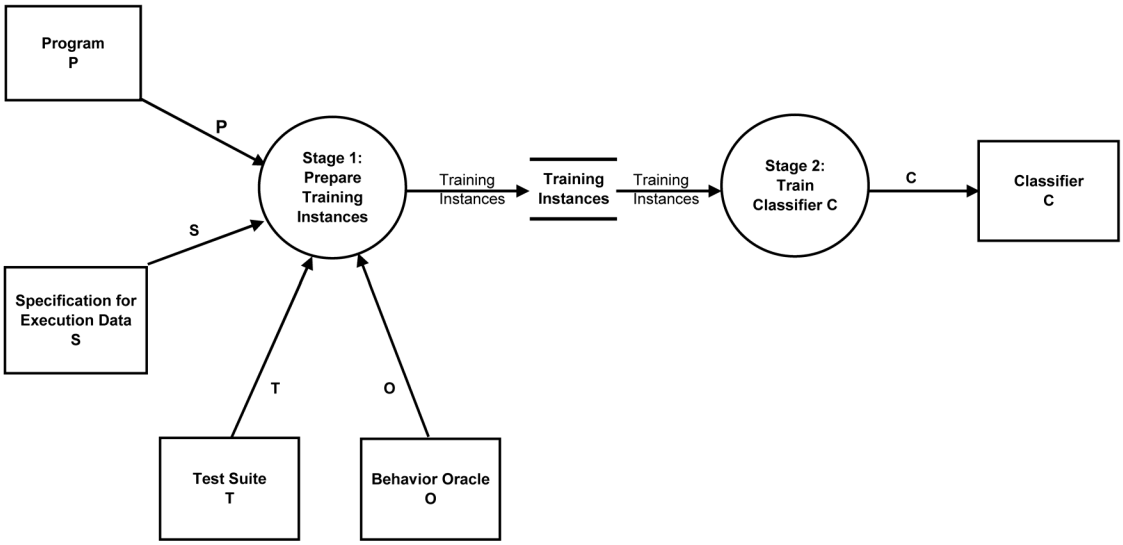


Figure 3.1. Overview of two-stage technique to build classifier.

Reading from left to right in Figure 3.1, the process labeled "Stage 1: Prepare Training Instances" takes as inputs a subject program *P* and a specification for execution data *S*. The program *P* is represented by its source code and the specification *S* identifies the data to be collected and the statistical model to be used, for example branch profiles modeled as a stochastic process. The output of Stage 1 is a database of training instances as shown in the center of Figure 3.1. This database in turn becomes the input for the second process labeled "Stage 2: Train Classifier." The output of Stage 2 is a classifier C for the program *P*.

Figure 3.2 is a dataflow diagram of the details of Stage 1 and Figure 3.3 is a dataflow diagram of the details of Stage 2. In Figure 3.2, Stage 1 is composed of three internal processes. Process I instruments *P* to collect the data specified by *S* and produces an instrumented version of *P*, named $\hat{P}$. Process II takes as input a test suite *T*

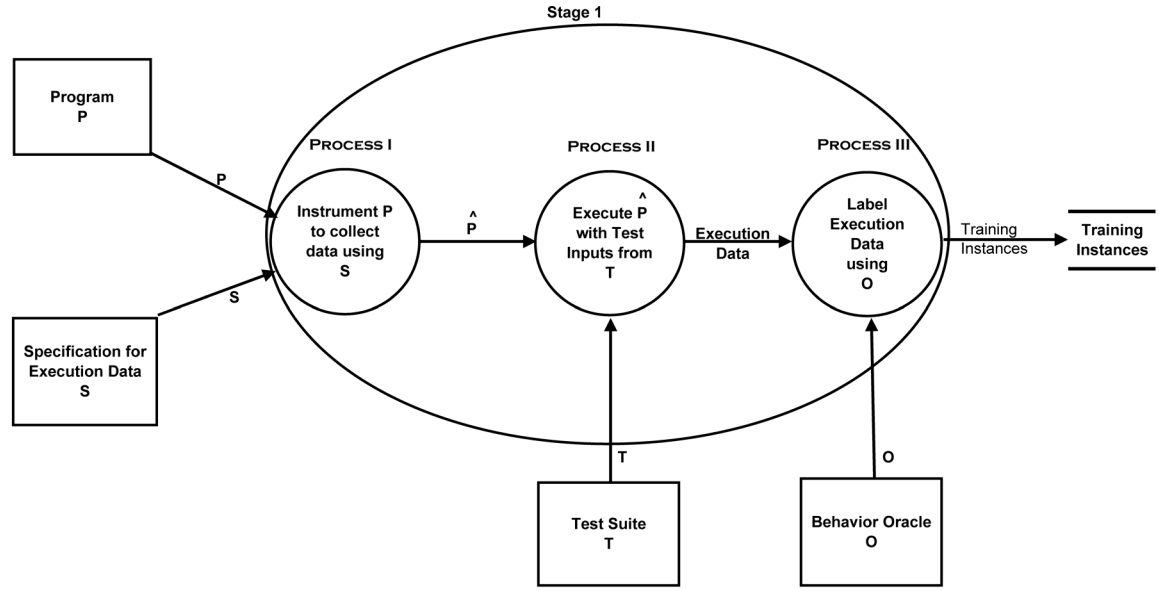


Figure 3.2. Stage 1: Prepare training instances.

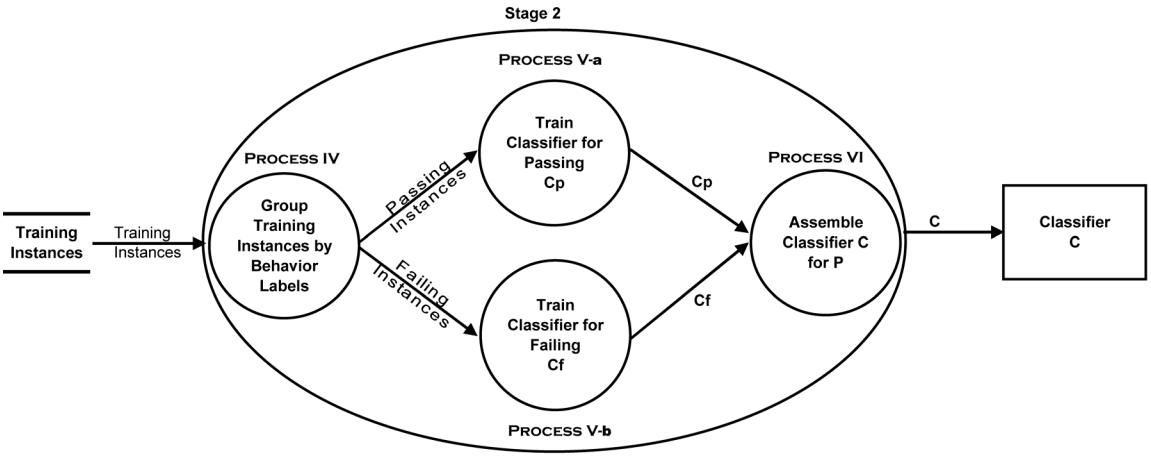


Figure 3.3. Stage 2: Train classifier.

and executes $\hat{P}$ once for each test input specified in *T*. The result of Process II is a collection of execution data produced by $\hat{P}$ and modeled according to *S*, and these modeled data flow into Process III. The external input to Process III is a behavior oracle *O*, which provides the correct label of "passing" or "failing" for the execution data produced by each execution of $\hat{P}$. The assignment of these labels is generally done by hand and thus incurs the expense of human labor. The output of Process III is a database of labeled and modeled execution data that will serve as the training data for the classifier.

Figure 3.3 depicts the details of Stage 2 of the technique as a dataflow diagram. The database on the left in Figure 3.3 is the database produced at the end of Stage I as shown in Figure 3.2. Process IV groups the training instances by their labels—in this case, "passing" and "failing." This grouping is a simple application of *supervised learning*, a machine-learning technique where the class labels for data items are known [35]. Supervised learning creates a model that maps data to its known labels. In this case, knowledge about the executions' two external behaviors is used to segregate the training data prior to classifier construction.

Next, each of Processes V-a and V-b automatically builds a classifier for its respective training set. Within each of these two processes, the classifier construction is via *unsupervised learning*, where the class labels that represent sub-behaviors within the principal behavior (i.e., "passing" or "failing") are not known. For example, the execution data labeled "passing" may include various sub-behaviors of "passing" depending on the inputs to the program. The goal of constructing a classifier *Cp* for the passing group is to obtain a useful estimate of the underlying characteristics of the

members of the passing group by finding patterns in the data. In this dissertation, the classifier-learning technique clusters data using an agglomerative hierarchical clustering algorithm, presented in the next section.

Process VI combines the resultant constituent classifiers into a single classifier *C*. This classifier is a collection of the individual classifiers trained in the previous processes, and classifier *C* is the output of this technique. Note that the complete technique uses batch learning as described in Section 2.3.1 to build the classifier *C*.

3.3 Agglomerative Hierarchical Clustering

As described above, the training data with which to build a classifier are the models that summarize the data specified and collected from a set of program executions as illustrated in Figure 3.1. The models used in this research are specified in Chapter 4.

One goal of classifier construction is to generalize the features of the models in order to find patterns in them that map to the behavior labels. An automated clustering algorithm that clusters the constituent models into a smaller number of representative models or patterns is a common way to reach this goal. One approach to discovering such clusters or patterns of models is to define a *similarity function* that measures some distance between two models. The specification of the similarity function is typically a heuristic learned by the researcher from the application domain. Clusters of models then emerge because their members are near to each other in terms of this metric. Each such cluster is then reduced to a single model that summarizes its constituent members. The trade-off here is a loss of precision—the individual executions are elided within the

cluster's model—for a gain in effectiveness resulting from fewer models. This tradeoff is a basic concern in cluster analysis and classifier design:

> One of the most important areas of research in statistical pattern classification is determining how to adjust the complexity of the model—not so simple that it cannot explain the differences between the categories, yet not so complex as to give poor classification on novel patterns [35].

The classifier-training paradigm used in this research is an adaptation of an established technique known as *agglomerative hierarchical clustering* [35]. With this technique, each training instance initially defines a cluster of size one. The technique proceeds iteratively by finding the two clusters that are nearest to each other according to some similarity function. These two nearest clusters merge into one cluster, and the technique repeats. The stopping condition for this clustering technique is either a fixed number of clusters or some valuation of the quality of the remaining clusters. The specific implementation of agglomerative hierarchical clustering used in this research is presented in Chapter 4.

3.3.1 Similarity functions for agglomerative clustering

As discussed in the previous section, a similarity function provides a numerical distance between two data models or clusters of models. There are three commonly used measures. The *nearest-neighbor algorithm* calculates the distance between two clusters by finding the minimum distance between two data points in the two clusters (i.e. the two nearest-neighbor data points.) When this algorithm is used to define the similarity measure, it eventually produces a minimal spanning tree of all the data points. It is well-

recognized that this algorithm can be too sensitive to outliers, thus producing a chaining effect that can yield poorly formed clusters [35]. An alternative measure that is less sensitive to outliers calculates the average distance between all points in the two clusters. This approach is computationally expensive, as it requires repeated pairwise calculations between all points in each cluster at each iteration of the agglomeration algorithm. A less expensive alternative measure is to calculate the distance between the centroids—the average of a cluster's members—of two clusters. This research uses this centroid calculation to reduce computation time and to decrease the sensitivity to outliers.

3.3.2 Description and complexity

The agglomerative hierarchical clustering algorithm begins with *n* data elements and a similarity function defined on those data elements. At the first iteration, there are $\frac{n*(n-1)}{2}$ pairwise similarity calculations made to detect the closest two data elements. Thus, in this first step of the algorithm, both the storage use and computation time are bounded by n^2. At each subsequent iteration, *n* is decremented by 1 as two data elements merge and therefore *n-2* new similarity calculations are required between the merged cluster and the remaining data elements. Assuming that the stopping criterion is $n = 1$, then there is limit of

$$\sum_{m=n-2}^{1} m = \frac{m*(m+1)}{2} = \frac{(n-2)*(n-1)}{2}$$

additional similarity comparisons to make until the algorithm stops with one cluster. Because the space and time requirements of both the initialization step and the remaining steps are bounded by n^2, the complexity of this algorithm is $O(n^2)$.

3.3.3 Automated classifier refinement

As described in Chapter 2, there are, in addition to the batch-learning technique, a number of learning strategies for training classifiers. This research concentrates on exploring the advantages of active learning [27], described in Section 2.3.1. Active learning provides a way for a classifier to refine itself adaptively after initially training with a batch of data. Active learning employs an interactive approach that can be used to control the costs of training classifiers because the classifier incorporates only those data elements that extend its range and ignores those data elements it recognizes. Thus, active learning is especially well suited to environments where the scope of the data is not yet fully known, as is generally the case with internal software behaviors. While an initial set of training instances might be drawn from the software testing process, for instance, exhaustive testing is usually not practical, if even possible. Consequently, software engineers do not have advance knowledge of specific unobserved internal behaviors. When these new behaviors do occur, an active-learning environment provides for qualitative feedback from an oracle, in this case the test engineer. This feedback allows for *refinement* of the classifier to include the new behavior. However, when the classifier recognizes new instances of known behaviors, these behaviors do not need hand-evaluation, and this is the source of the cost savings.

4 SELECTION OF FEATURES AND THEIR MODELS

This chapter presents the approach used in this research to select features of program execution data and the approach used to model these features.

The conventional software engineering approach to identifying and validating internal program behaviors views each execution as the transformation of inputs into outputs. This transformation can be monitored and verified at every statement, as for example when engineers test a program for statement coverage by inserting probes at each statement [15]. The input space can be partitioned and the program executed with exemplar inputs for each partition. Finally, an observer can confirm whether the program produces the correct outcome for a given input. For example, consider the procedure *TriangleType* presented previously in Figure 1.1 and with its CFG in Figure 2.1. A visual analysis shows that there are four possible paths through the procedure, each corresponding to one of the possible triangle types:

{entry, S1, S2, S8, exit} : not a triangle;

{entry, S1, S2, S3, S4, S8, exit}: equilateral;

{entry, S1, S2, S3, S5, S6, S8, exit}: scalene;

{entry, S1, S2, S3, S5, S7, S8, exit}: isosceles.

By considering the CFG of *TriangleType* as a directed graph with unweighted edges, then each of these four paths is equally likely to be traversed to reach *Exit* from *Entry*. However, if the distribution of the input space is considered, then the path {*entry, S1, S2, S8, exit*} corresponding to "not triangle" is the most likely path. This is because for any two non-zero natural numbers representing two sides of a potential triangle, the

"legal" value of the integer representing the third side comes from a bounded and finite set whose largest member is the sum of the length of the two given sides minus 1 and whose smallest member is 1. Any of the infinite number of integers not in this set is an "illegal" third side. Although this is a simple case, it is clear that such an analysis performed without regard to the actual use of the program might lead a development team to concentrate on optimizing this most probable path.

An alternative approach is to suppose that the users of this program are measuring millions of actual triangles in the field and that they seek to automate the labeling of the triangle type. The developers will not be able to analyze economically each execution to detect whether it correctly computed the triangle type. The developers may even anticipate a different mix of triangle types than actually occur, because in this case, none of the inputs will be illegal triangles. However, they may be able to efficiently monitor certain features of the internal behaviors of the program and then determine statistically the distribution of triangles that users tested and hence the distribution of CFG paths taken. The developers could then tailor their optimization efforts to the reported usages of the program. They could also measure shifts in the relative frequencies of the three types of triangles tested over time. This second, alternative, approach is empirical and statistical.

4.1 Research approach

The alternative approach that uses statistical and probabilistic techniques enables the design and implementation of cost-effective behavior recognition. In the limit, both the external and internal behaviors of a particular software program can be characterized by examining the collection of all its executions, embodying the diversity of all users,

inputs, and execution environments for all copies of the program. This collection of executions is unpredictable and generally unbounded, as in the case of the procedure *TriangleType*, and these properties invite a data-driven approach to analysis in contrast to an analysis that requires an enumeration of each execution. The *data-driven approach* used in this research selects specific features of program executions, collects data from probes of these features in an instrumented program, summarizes and models these data, and then evaluates the aggregated data for its ability to support effective behavior classifiers. In particular, the approach uses machine-learning techniques to build probabilistic models of the data collected from program executions. Probability theory has a distinguished history in the context of modeling processes, and it is natural to explore its use in the service of software engineering:

> The modern approach to stochastic modeling is to divorce the definition of probability from any particular type of application. Probability theory is an axiomatic structure, a part of pure mathematics. Its use in modeling stochastic phenomena is part of the broader realm of science and parallels the use of other branches of mathematics in modeling deterministic phenomena [94].

The approach in this research is thus to model data collected about selected events in a program's execution as a random or stochastic process and then to map these data models of internal program behaviors to external behaviors. For example, a program cannot in general detect whether its own outcome is correct or not (i.e., whether the execution passes or fails). However, with a probabilistic approach, this research demonstrates that stochastic models of certain events in an execution can map to the two external program behaviors labeled "passing" and "failing." Furthermore, this research

shows that behavior classifiers that express this mapping can reliably predict these external behaviors for selected subject programs.

This alternative to conventional analysis is above all a modeling choice. The design of models is inherently as complex as any software engineering task that seeks parsimonious descriptions of program behaviors. There is no best model for a set of events or a behavior; rather, a model is judged by its *usefulness*, including the cost of its creation [58]. If a model effectively detects and predicts program behaviors, then it is a useful model.[9]

4.1.1 Probabilistic reasoning

A discrete-time stochastic process is a sequence of random variables. Let $t = 0, 1, 2, 3, ...$ be the time at which we observe a random variable X_t that represents the state of a system. Then the sequence $\{X_0, X_1, X_2, ...\}$ can be written as $\{X_t, t >= 0\}$ and is called a stochastic process. Each X_t can take a set of values S, which is the *state space* of the process $\{X_t, t >= 0\}$. A ubiquitous example of this definition is X_t = the value of the toss of a six-sided die on throw number t. Here, $S = \{1, 2, 3, 4, 5, 6\}$. In the case of the example procedure *TriangleType*, shown in Figure 2.1, X_t can be defined as the statement number that is executing at time t. Then $S = \{entry, S1, S2, S3, S4, S5, S6, S7, S8, exit\}$.

[9] My inspiration for exploring this approach comes from several sources. During my research into Software Tomography, I became intrigued with the branching behavior of executing programs [21]. Software Tomography is a technique used in monitoring deployed software wherein a particular task, such as detecting branch coverage, is divided into subtasks. While watching visualizations for hours on end during simulations of tomography, I was intrigued by the patterns of branch activity that appeared. In particular, I wondered whether there was a way to recognize these patterns mathematically. I studied probabilistic graphical models and related research problems in computer vision and speech recognition. I found useful tools from the field of speech recognition [75]. I saw a parallel between estimating an utterance from an acoustic waveform and estimating the external behavior of a program from dynamic-analysis data. I learned that these relationships are inherently stochastic, due to the richness and variability of the measurement set. My first strategy was to leverage the successful tools and methods of speech recognition in this novel context of program behavior analysis.

Note that this definition requires that the procedure is executing and without making any assumptions about the passage of time.

4.1.2 Discrete-time Markov chains

One special class of stochastic processes is the discrete-time Markov chain. Consider the above-described stochastic process for procedure *TriangleType*. Assume that at time $t = 4$ the observed values of X_0, X_1, X_2, X_3, X_4 are *entry, S1, S2, S3, S5*. Is it possible to predict with a probability what the state of the process will be at $t = 5$? One answer is that X_5 depends on X_0, X_1, X_2, X_3, X_4 and the input parameters *i, j, k*. However, there may be a very large set of conditional relationships between the random variables and their states. These conditional relationships exist because of the particular ordering of the state-values of the random variable *X*. Markov in the late 1800s proposed a simplification where to predict X_5, all that is required is X_4. If the stochastic process has this property at every time $t > 0$, then it has a *Markov property*. Alternatively, the *Markov assumption* is that the next state in the modeled state space depends only on the current state and not on any previous state. This simplifying assumption removes the conditional relationships represented by a sequence of states.

Thus, a stochastic process $\{X_t, t >= 0\}$ with state space *S* is called a Discrete-time Markov Chain (DTMC) if, for *x* in *S*:

$$P(X_{n+1} = x \mid X_0, X_1, X_2, ..., X_n) = P(X_{n+1} = x \mid X_n).$$

A further simplifying assumption is that the DTMC is homogeneous with respect to time (i.e., that it is a *stationary process*—a process governed by rules that do not change over

time [94].) The types of software programs studied in this research are stationary processes because the program code does not change between executions.

4.2 Stochastic features of program execution data

This research focuses on modeling the data collected from a program's execution that are then studied for patterns that map to characterizations of the external behaviors of the program. The term feature used in this research has a specific meaning in statistical pattern classification and machine learning. A *feature* is one of a set of characteristics of the data being studied [35]. This sense of the term *feature* is used in software engineering research in automated classification as well (e.g., [20, 23, 32, 79]). The class of data features modeled in this research captures the frequency of transitions among certain statements in a program during the program's execution.

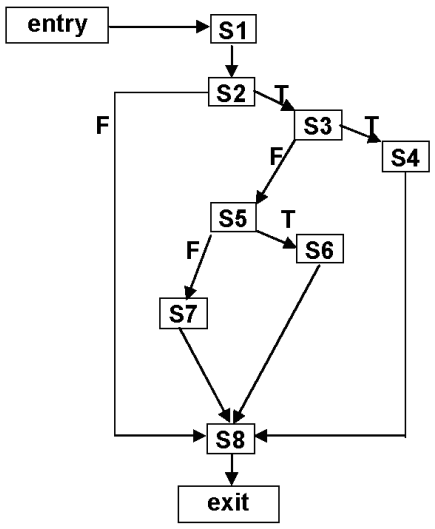
The approach taken is to collect the profile data from an individual execution and reduce it immediately to a DTMC through normalization of the rows of the transition matrix, as explained below. This approach contrasts with the conventional approach of performing the reduction to a model near the end of the execution-clustering process (e.g., [32, 79, 78].) The motivation for this is three-fold. First, in the speech-recognition approach that inspires this work, Markov models summarizing speech data are the basic elements used in classifier training [82]. Second, using the same representation of the data at every stage of the clustering process provides for a general approach to classification at any stage of the clustering process, as detailed in Section 4.4.3. Third, the DTMCs preserve most of the information within the transition matrices that represent the collected profile data, losing only the absolute profiles.

For example, one feature used in this research is the branch profile. A *branch* is an edge between a predicate (source) statement and a target (sink) statement in the CFG of a program. There are benefits to using a DTMC to model the branching behavior of an executing program. First, the state space, *S*, is finite and is composed of the predicate statements plus the target statements of each predicate statement in the program. Secondly, this model has the same state space *S* for every execution of the program. The conventional representation of a DTMC is a two-dimensional matrix *M* where each dimension is an enumeration of the state space. The storage requirement of this representation is thus n^2, where $n=|S|$. Each cell of the matrix M is identified by its row and column: $M(row, column)$. Both *row* and *column* are members of *S*. Thus, cell $M(row, column)$ contains the transition probability from the *row* state to the *column* state. The predicates in a program can branch on "true" or "false" or may be more complex and have more than two outcomes, such as when a *switch* construct is used in the C language. These more complex predicates can be reduced to a series of "true" and "false" branches with the use of "if-then" constructs that replace, for example, each *case* statement in the *switch* construct. In graph theory, this transformation is the same as the equivalence between *n-ary* and *binary* trees. In other words, a *switch* statement can be represented either as a *n-ary* tree or a *binary* tree. When every predicate is represented as a binary tree, then the number of possible branch transitions in the DTMC is $2*n$, where n is the number of binary predicates. An adjacency matrix representing the CFG of a procedure will contain "1"s for these possible transitions and "0"s otherwise. For example, the CFG for the procedure *TriangleType* is shown in Figure 4.1 on the left and the adjacency matrix representation of this CFG is shown on the right. The nominal size of the

adjacency matrix is $10^2 = 100$, but the count of possible transitions—shown as "1"s—is only 12. Thus, in this case the adjacency matrix can be stored as a sparse matrix.[10]

The DTMC that models the branches in *TriangleType* will therefore also be a sparse matrix with the state space *S = {entry, S1, S2, S3, S4, S5, S6, S7, S8, exit}*. By convention, the entry node is included to denote the initial state of the model and the exit node is included as an absorbing state that, once reached, allows no further transitions to other states. The DTMC will have non-zero entries showing the probabilities of possible state transitions and "0"s for impossible transitions. Static analysis shows that these impossible transitions will never occur, as for example from *S1* to *Entry*.

The DTMC derives its state space and possible transitions from the CFG and its adjacency matrix. However, in this research, the entries in the DTMC derive from the



	entry	S1	S2	S3	S4	S5	S6	S7	S8	exit
entry	0	**1**	0	0	0	0	0	0	0	0
S1	0	0	**1**	0	0	0	0	0	0	0
S2	0	0	0	**1**	0	0	0	0	**1**	0
S3	0	0	0	0	**1**	**1**	0	0	0	0
S4	0	0	0	0	0	0	0	0	**1**	0
S5	0	0	0	0	0	0	**1**	**1**	0	0
S6	0	0	0	0	0	0	0	0	**1**	0
S7	0	0	0	0	0	0	0	0	**1**	0
S8	0	0	0	0	0	0	0	0	0	**1**
exit	0	0	0	0	0	0	0	0	0	0

Figure 4.1. TriangleType CFG and adjacency matrix.

[10] A sparse matrix is a matrix populated primarily with zeros that can be stored more efficiently than a rectangular matrix by various means as for example, an edge list.

data collected about these transitions during an execution of the program. The Markov assumption here is at odds with the conventional view of state transitions within a program. Clearly, there are conditional dependencies on previous statements. However, this modeling assumption is made primarily for reasons of tractability, as discussed in Section 4.1.2. This simplification of the state transitions permits the use of first-order Markov models and the empirical studies in Chapter 7 demonstrate the efficacy of this modeling choice.

Each cell in the DTMC contains the probability that the transition from the state in the row to the state in the column occurred during the execution. These probabilities are facts about the particular execution of the program and do not describe the program itself. As a simple example, consider the predicate at node *S2* in Figure 4.1. When this predicate evaluates as "true," control in the execution flows to node *S3* and when it evaluates as "false," control flows to node *S8*. These two branches are the only possible transitions from state *S2*, as shown in the adjacency matrix in Figure 3.1, and the associated transition probabilities are $P(S3 \mid S2)$ and $P(S8 \mid S2)$. Because these two transitions are mutually exclusive and exhaustive (the only possible transitions from *S2*), then the axioms of probability state that

$$P(S3 \mid S2) + P(S8 \mid S2) = 1.$$

This means that if during an execution, the predicate *S2* is never reached, then

$$P(S3 \mid S2) = P(S8 \mid S2) = 0.5.$$

The interpretation is that either transition was equally possible during the execution, even though neither occurred. This convention is a standard mathematical transformation when working with Markov models in this context [82]. The DTMC will be used to

calculate and compare probabilities. If each of these two entries were zero instead of 0.5, then the multiplicative effect would be to reduce to zero any probability calculated as passing through the transition represented by the entry. By normalizing the two entries to 0.5, each has the same multiplicative effect on any probability calculation.

As a practical matter, the way these two probabilities are calculated from execution data is straightforward. For example, if the procedure *TriangleType* were called from within procedure *Main* 15 times and if the branch from *S2* to *S3* occurred 5 times and the branch from *S2* to *S8* occurred 10 times during this execution, then the two probabilities are calculated as follows:

$$P(S3 \mid S2) = 5/15 = 1/3 \text{ and } P(S8 \mid S2) = 10/15 = 2/3.$$

A technical problem arises when a possible branch transition does not occur during an execution. The issue is that the transition had a very small, but non-zero, probability of occurring. If this transition were assigned a zero in the DTMC modeling the execution, then the interpretation is that the transition is impossible. However, since the transition is possible, the convention is to assign the transition a very small probability, and then to recalculate the probabilities for that row so that they sum to a total of one, as required by the axioms of probability. For example, if the branch from *S2* to *S3* occurred 15 times and the branch from *S2* to *S8* did not occur during this execution, then the initial calculation yields:

$$P(S3 \mid S2) = 15/15 = 1 \text{ and } P(S8 \mid S2) = 0/15 = 0.$$

To remove this zero probability, it is replaced by setting $P(S8 \mid S2)$ = 0.0001, for instance, and then normalizing the probabilities so that they add to one:

$$P(S3 \mid S2) = 1/1.0001 = 0.9999 \text{ and } P(S8 \mid S2) = 0.0001/1.0001 = 0.0001.$$

In the case where neither transition occurred during the execution, each zero if replaced by 0.0001 and the results normalized so that they add to one:

$$P(S3 \mid S2) = 0.0001/0.0002 = 0.5 \text{ and } P(S8 \mid S2) = 0.0001/0.0002 = 0.5.$$

The interpretation of this substitution is that during the execution, even though neither branch executed, each had an equal probability of executing. As explained above in this section, this artifice prevents the introduction of zeroes into any probability calculation involving transitions from statement *S2*.

4.3 Automatically building Markov models from execution data

The basic procedure for building a DTMC from the transition data collected during a single execution is performed in four steps:

(1) Create a transition matrix that contains the actual measured count of each transition, also known as the profile or frequency of the transition.

(2) Normalize each row so that the cell entries become probabilities and each row sums to a probability of one, as described above.

(3) Assign a very small probability such as 0.0001 to any possible transitions that did not occur during the execution (i.e., their transition profiles or counts are zero). This procedure prevents the occurrence of probabilities of zero that cancel probability calculations using this DTMC.

(4) Re-normalize the rows so that they each sum to a probability of one. The DTMC now contains a non-zero probability for every possible transition and a zero for all impossible transitions.

The algorithm BUILDMODEL, shown in Figure 4.2, is an implementation of this procedure. BUILDMODEL constructs a matrix representation of a DTMC from state-transition profiles. Note that as explained above, the square matrix representation is used for illustration, but in an implementation, efficiency dictates the use of sparse matrices. BUILDMODEL has three inputs: *S*, *D*, and *b*. *S* is the set of states used to specify the transitions. *D* contains the data collected from a program execution, consisting of ordered triples and including an entry for every possible transition, including those with profiles, or counts, of zero. Each ordered triple describes a transition from a state s_{from} to a state s_{to} with the corresponding transition profile or frequency recorded during the execution:

$$(s_{from}, s_{to}, profile).$$

Symbol *b* is the behavior label for the model, such as "passing" or "failing." The output of BUILDMODEL is the triple (*M, D, b*) of the model, the input profile data, and the behavior label.

The algorithm BUILDMODEL operates as follows. In line 1, the algorithm initializes the matrix *M* for the model using the cardinality of *S*. In lines 2-3, each transition in *D* that involves states in *S* is recorded in *M*. In lines 4-8 each row in matrix *M* is normalized by dividing each element in the row by the sum of the elements in the row, unless the sum is zero. In lines 9-11, each possible transition that still has a 0 probability is changed to a small probability of 0.0001. Then in lines 12-16, the model is normalized again to produce rows that sum to probability 1. As an example, consider the single predicate illustration from Section 4.1. Suppose that the branch from *S2* to *S3*

Algorithm BUILDMODEL *(S, D, b)*

Input: $S = \{s_1, s_2, \ldots, s_n\}$, a set of states

$D = ((s_{from}, s_{to}, profile), \ldots)$, a list of ordered triples

$b =$ a string representing a behavior label

Declare: M, an array of floating point numbers to store the DTMC

Output: (M)

(1) $M \leftarrow new\ float\ Array[|S|, |S|]$, initialized to 0s

(2) **foreach** $(s_{from}, s_{to}, profile) \in D,\ where\ s \in S$

(3) $\quad M[s_{from}, s_{to}] \leftarrow M[s_{from}, s_{to}] + profile$

(4) **for** $i \leftarrow 0$ **to** $(|S| - 1)$

(5) $\quad rowSum \leftarrow \sum_{j=0}^{|S|-1} M[i, j]$

(6) $\quad$ **if** $(rowsum > 0)$

(7) $\quad\quad$ **for** $j \leftarrow 0$ **to** $(|S| - 1)$

(8) $\quad\quad\quad M[i, j] \leftarrow M[i, j] / rowSum$

(9) **foreach** $(s_{from}, s_{to}, profile) \in D,\ where\ s \in S$

(10) $\quad$ **if (** $M[s_{from}, s_{to}] == 0$ **)**

(11) $\quad\quad M[s_{from}, s_{to}] \leftarrow M[s_{from}, s_{to}] + 0.0001$

(12) **for** $i \leftarrow 0$ **to** $(|S| - 1)$

(13) $\quad rowSum \leftarrow \sum_{j=0}^{|S|-1} M[i, j]$

(14) $\quad$ **if** $(rowsum > 0)$

(15) $\quad\quad$ **for** $j \leftarrow 0$ **to** $(|S| - 1)$

(16) $\quad\quad\quad M[i, j] \leftarrow M[i, j] / rowSum$

(17) **return** (M)

Figure 4.2. Algorithm BuildModel.

occurred 5 times and the branch from *S2* to *S8* occurred 10 times during a "passing" execution. The inputs to BUILDMODEL would be:

- $S = \{S2, S3, S8\}$
- $D = ((S2, S3, 5), (S2, S8, 10))$
- *b = "passing."*

In this case, the output *M* is the DTMC of this example execution with two non-zero entries corresponding to (as calculated in Section 4.2):

$$P(S3 \mid S2) = 5/15 = 1/3 \text{ and } P(S8 \mid S2) = 10/15 = 2/3.$$

The complexity of BUILDMODEL is directly related to the size of the state space *S*, denoted as *n*. The model *M* that is the output of the algorithm is bounded in size by n^2. The model *M* is accessed cell by cell at each of the five stages of the algorithm. Thus, the time spent processing *M* is bounded by $5 * n^2$. The algorithm's time and space complexity is therefore $O(n^2)$.

4.4 Automating behavior classification with DTMCs

A behavior classifier is a map from an execution feature, such as branch profiles, to a label for program behavior, such as "passing" or "failing," as described in Chapter 3. This research has addressed the three tasks of classifier design: (1) identifying a set of features, (2) selecting a classifier architecture, and (3) implementing a learning technique for training the classifier using labeled data [69]. This section presents the details of how all three tasks work together to produce the automated classification of behaviors.

4.4.1 Clustering DTMCs

The similarity function used in this research is the distance between the centroids of two clusters. When the agglomerative hierarchical clustering algorithm begins (described in Section 3.3), each data point is a cluster of size one. The data points are themselves the DTMCs created by the BUILDMODEL algorithm. Any matrix and thus a DTMC can be represented as a vector. There are a number of ways to then define the distance between any two DTMCs, *M1* and *M2*, whose vectors are *m1* and *m2*, for example:

(1) absolute vector difference = $|m1 - m2|$;

(2) vector distance = $||m1 - m2||$, where $||v|| = \sqrt{v * v}$;

(3) cosine of angle between *m1* and *m2* = $m1 \bullet m2 \,/\, (||m1|| * ||m2||)$;

(4) Hamming distance between *m1* and *m2*; The *Hamming distance* between two binary numbers is the count of bit positions in which they differ. This is a variation on (1) where the vectors are transformed to bit vectors. Since each cell of a DTMC contains a value between zero and one, this transformation is done by using a heuristically-determined threshold *t* such that any member of *m1* or *m2* with a value less than or equal to *t* becomes a zero and any value between *t* and one becomes a one.

This research investigated both the absolute vector difference and the Hamming distance. In the empirical results presented in Chapter 7, each is used in a specific context where it proved most beneficial. Once the agglomerative clustering algorithm has chosen two clusters as having the closest means, it merges the two clusters. For DTMCs this process is straightforward. The two transition matrices from which these

two DTMCs were derived are added together cell by cell, a process of matrix addition. The resultant merged transition matrix is then row-normalized using the procedure described in the algorithm BUILDMODEL to create the merged DTMC. This DTMC has the same size and state space as its components and its cells contain the transition probabilities representing the mean of its component data points.

4.4.2 Agglomerative hierarchical clustering with DTMCs

The algorithm TRAINCLASSIFIER, shown in Figure 4.3, trains a classifier from a collection of models generated by BUILDMODEL. TRAINCLASSIFIER also implements agglomerative hierarchical clustering. TRAINCLASSIFIER has three inputs: *S*, *T*, and *SIM*. *S* is a set of states that identify the event transitions for calls to BUILDMODEL. *T* is a list of pairs each summarizing one execution *i* and containing a data structure *D* as defined in BUILDMODEL, and a behavior label b_l. There can be *k* behaviors but in this research, $k = 2$, for "passing" and "failing." *SIM* is one of the similarity functions discussed above that takes two DTMCs as arguments, and returns a real number that is the computed difference between the models. TRAINCLASSIFIER outputs a classifier *C* that is a set of the clusters, each represented by a DTMC, that remain at the end of the clustering process. Initially, the classifier *C* contains clusters that are the singleton data points. The agglomerative clustering process reduces the cardinality of *C*. As discussed earlier, this clustering trades a loss of specificity for a gain in efficiency resulting from fewer component DTMCs.

Algorithm TRAINCLASSIFIER *(S, T, SIM)*

Input: $S = \{s_0, s_1, \ldots, s_n\}$, a set of states

$T = ((D, b_l)_i, \ldots)$, training set as a list of ordered pairs, $i - 0,1,2...$

where $D = ((s_{from}, s_{to}, profile), \ldots)$,

b_l = a "behavior label", $0 < l <= k$, *the count of behavior groups*

SIM, a function to compute the similarity of two Markov models

Output: *C*, a set of Markov models, initially empty

(1) **foreach** $b_l, 0 < l <= k$

(2) $C_{b_l} \leftarrow \{\}$, declare a Classifier for behavior b_l as an empty set

(3) **foreach** C_{b_l}, $0 < l <= k$

(4) **foreach** $(D, b_l)_i \in T, i = 0,1,2,...$

(5) $C_{b_l} \leftarrow C_{b_l} \bigcup (BuildModel(S, D, b_l), D)$

(6) $Deltas \leftarrow \{\}$, an empty set to collect pairwise deltas

(7) $Stats \leftarrow new\ Array[|C_{b_l}|]$, clustering statistics

(8) **while** $(|C_{b_l}| > 2)$

(9) // agglomerative hierarchical clustering

(10) **foreach** $(M, D)_i \in C_{b_l}, 0 < i < |C_{b_l}|$

(11) **foreach** $(M, D)_j \in C_{b_l}, i < j <= |C_{b_l}|$

(12) $Deltas \leftarrow Deltas \bigcup SIM(M_i, M_j)$

(13) $Stats[|C_{b_l}|] \leftarrow StdDeviation(Deltas)$

(14) **if** *Knee(Stats)* **then break**

(15) **else**

(16) M_x, M_y *chosen as closest in Deltas*

(17) $D_{merged} \leftarrow D_x \bigcup D_y$

(18) $M_{merged} \leftarrow BuildModel(S, D_{merged}, b_l)$

(19) $C_{b_l} \leftarrow (C_{b_l} - M_x - M_y) \bigcup M_{merged}$

(20) $C \leftarrow C \bigcup C_{b_l}$ add Classifier for behavior l to C

(21) **return** *C*

Figure 4.3. Algorithm TrainClassifier.

In lines 1-2, TRAINCLASSIFIER initializes an empty classifier C_{b_l} for each discrete behavior b_l. In this research, these labels are "passing" and "failing." Line 3 begins the processing loop for each behavior b_l. In lines 4-5, the algorithm adds an ordered pair *(M,D)* to the appropriate classifier C_{b_l} based on the b_l in *(D,* b_l*)*. In this pair, *M* is the output of BUILDMODEL*(S,D,*b_l*)*. After completing this loop, C_{b_l} contains one DTMC for each training instance with the behavior label b_l. Lines 6-7 initialize *Deltas* and *Stats*, two data stores that are explained below.

The remainder of the algorithm clusters the models in each C_{b_l} to reduce their populations. The agglomerative clustering process begins after line 8. Line 8 establishes the default stopping criterion as two clusters, in the event that the stopping heuristic *Knee* does not detect a deterioration at line 14. This could occur for example if all the data points were equidistant. In lines 10-12, *SIM* calculates the each pairwise difference and *Deltas* accumulates them at line 12. At each iteration, the algorithm calculates the standard deviation for the values in *Deltas* and stores it in *Stats* [$|C_{b_l}|$] at line 13. Because the cardinality of C_{b_l} decreases by one per iteration, it serves as an index into *Stats* []. The function *Knee* implements a stopping heuristic. *Knee* checks the set of standard deviations accumulating in *Stats* [] at line 14 to determine the rate of change in the slope of a line fit to the values in *Stats* []. *Knee* is detecting a "knee shape," or sharp bend, in the graph of the standard deviations in *Stats* []. A graph-plotting program, for instance, could aid a manual detection of this knee, but *Knee* provides an automatic detector for use in the empirical evaluations conducted in this research. *Knee* detects a "knee" when the sum of standard errors (SSE) in a linear regression of the data points in

Stats [] increases by a factor of ten or more from its value in the previous iteration. If a knee is detected, the clustering stops for that behavior group, and the models in C_{b_l} are added to *C*, the final classifier. In the absence of a knee, the process stops with two models, per the constraint in line 8. Otherwise, the algorithm merges the two closest models M_x and M_y in lines 16-19, by calling BUILDMODEL with the union of the corresponding profile sets D_x and D_y. At line 20, the classifier *C* incorporates the clustered models in C_{b_k}. After processing all the behavior groups, the algorithm returns the final *C*.

The use of a stopping criterion for the clustering process must be determined heuristically by the researcher using domain knowledge [35]. For the agglomerative hierarchical clustering algorithm to produce a single cluster is appropriate if only one class of data exists. If there is more than one class within the data points then there is a need for a stopping criterion. In the specific case of training a classifier for each of the behaviors "passing" and "failing," there are likely sub-behaviors that will naturally cluster within each of these. Consider the example procedure *TriangleType* for which the four possible internal behaviors are four paths in the CFG, shown on page 36. Each of these internal behaviors occurs during a "passing" behavior of the program, yet they have different properties. Using this domain knowledge, it is clear that agglomerative hierarchical clustering of a collection of training instances that included 100 of each of these four internal behaviors should yield four clusters, one representing each path. With this advance knowledge of the four internal behaviors, the stopping criterion is four. In this trivial case, clustering works perfectly to create four DTMCs, where each represents one of the internal behaviors. The reason is that the model for each internal behavior is

always the same, reflecting the single traversal of a path through the procedure. Consequently, any pair of DTMCs both representing one path will have a measured similarity distance of zero and will be merged. The merged model will be indistinguishable from either of its identical components. With a stopping criterion of four, each of the clusters then represents one of the four possible paths.

In the general case when the number of constituent sub-behaviors that exists in the data set is unknown, then a technique is needed to detect the appropriate stopping point for the clustering algorithm. This problem is known in the pattern-recognition community as "the problem of validity." [35] There are many theoretical and practical approaches available. The basic problem is in determining whether a clustering level with c clusters better describes the data than the clusters at level $c+1$. Because this research effort is not concerned with identifying the specific sub-behaviors that clusters represent, but rather in building a classifier that can reliably predict "passing" or "failing" behaviors, the decision boundary is flexible. The choice of c can be informed by statistical properties of the clustering process. Every practical solution to this problem is a heuristic learned from the data, and the one used in this research works well for the subjects investigated. This heuristic calculates the standard deviation of the pairwise distances between the clusters at each level. Thus, one measure exists for each iteration of the clustering algorithm. The heuristic then tracks the standard deviation measure across the iterations and monitors the quality of a linear regression of these measures up to the current iteration of the algorithm. When the regression results signal an upward trend in the standard deviations, this point is selected as the stopping criterion. Just before this deterioration, the clusters, represented by their means, are relatively evenly

distributed as measured by the standard deviation of their inter-cluster distances. The function *Knee* described above implements this heuristic.

The complexity of TRAINCLASSIFIER is directly related to both the size of the state space *S*, denoted as *n,* and the number of executions in the training set, denoted by *i*. The *k* behaviors serve to segregate the executions in the training set, but the total number of executions remains *i*. The initialization of the algorithm uses the algorithm BUILDMODEL for every execution. Since BUILDMODEL is bounded in space and time by n^2, this initialization is bounded in space and time by $i * n^2$. If *n* and *i* are of the same order of magnitude, then this bound is n^3. This is generally true of the subjects and classifiers used in the empirical studies presented in Chapter 7. The second part of TRAINCLASSIFIER is the implementation of the agglomerative hierarchical clustering algorithm presented in Section 3.3.2 and shown to be bounded in space and time by n^2. Therefore, the time and space complexity of TRAINCLASSIFIER is $O(n^3)$. However, for the features and subjects used in this research, the DTMC transition arrays are bounded by a small constant times *n*, and so this complexity is more nearly $O(n^2)$ in practice.

4.4.3 Automatic classification

The classifier *C* constructed by TRAINCLASSIFIER contains several clusters, each represented by a DTMC. The goal in this research is to instrument the subject program, collect the data, model a specific feature of the data as a DTMC, and then ask the classifier to classify the DTMC and thus label the execution. The procedure *TriangleType* has two external behaviors that will serve to illustrate this process. These

behaviors are “not triangle” and “isosceles,” and in this special example case, these are the external labels for two “passing” behaviors. The two “passing” behaviors of *TriangleType* will illustrate without loss of generality the supervised construction of a classifier.

Because there are only four possible paths through *TriangleType*, there are only four different possible DTMCs that will model executions that traverse these paths. Figure 4.4 shows the profile entries on the left and the resultant DTMCs on the right for typical executions of the two behaviors under consideration: *{entry, S1, S2, S8, exit}* : not a triangle and *{entry, S1, S2, S3, S5, S7, S8, exit}* : isosceles.

In Figure 4.4, the behavior labels for each row are placed vertically on the left. The left-hand matrix for each behavior shows the profile data for one execution of the procedure. In this simple example, the transition counts are “1,” as no looping or repeats occur. The arrowheads in the column to the left of the profile matrix mark the states that occur in the path for the behavior. The “0”s represent possible transitions in the CFG that were not exercised in the execution. The blank cells represent the impossible transitions. Note that the state space for this discussion is the set of all nodes in the CFG. The presence of transitions between nodes that occur without branching does not affect the results and makes the matrices more readable for this discussion. The matrix on the right in each row is the DTMC calculated from the transition matrix on the left in the same row. The DTMC rows are normalized as described in Section 4.2. For this example, to save space and for readability, the small probability is 0.01 in lieu of the 0.0001 used in BUILDMODEL. Note that when there is only one transition possible from a row state then

not triangle

	entry	S1	S2	S3	S4	S5	S6	S7	S8	exit
entry		1								
S1			1							
S2				0					1	
S3					0	0				
S4									0	
S5							0	0		
S6									0	
S7									0	
S8										1
exit										0

Profile

	entry	S1	S2	S3	S4	S5	S6	S7	S8	exit
entry		1.00								
S1			1.00							
S2				0.01					0.99	
S3					0.50	0.50				
S4									1.00	
S5							0.50	0.50		
S6									1.00	
S7									1.00	
S8										1.00
exit										1.00

DTMC-A

isosceles

	entry	S1	S2	S3	S4	S5	S6	S7	S8	exit
entry		1								
S1			1							
S2				1					1	
S3					0	1				
S4									0	
S5							0	1		
S6									0	
S7									1	
S8										1
exit										0

Profile

	entry	S1	S2	S3	S4	S5	S6	S7	S8	exit
entry		1.00								
S1			1.00							
S2				0.99					0.01	
S3					0.01	0.99				
S4									1.00	
S5							0.01	0.99		
S6									1.00	
S7									1.00	
S8										1.00
exit										1.00

DTMC-B

Figure 4.4. Profiles and calculated DTMCs for TriangleType.

the entry in the DTMC is a probability of one, as for example *{S8, exit}*. The presence of these singleton transitions in the model makes it more readable and has no effect whatever on the similarity measures, since they exist identically in all DTMCs derived from this model. Furthermore, in the probability calculations detailed in the next section, which involve products of the individual probabilities, a probability of one has no multiplicative effect.

The two DTMCs shown in Figure 4.4 are, for the purposes of this discussion, the component clusters of the classifier *C* built using TRAINCLASSIFIER with a stopping criterion of two, based on knowledge of the domain. To illustrate how classifier *C* will classify another execution of *TriangleType* that also exhibits behavior "isosceles," data is

collected from the execution. In this simple example, then, the path for the new execution is:

E = *{entry, S1, S2, S3, S5, S7, S8, exit}* : isosceles.

The classification process follows a voting protocol. Each of the constituent cluster models of C rates E with a probability score. The model in C with the highest (relative) probability score for E provides the behavior label for E. The *probability score* (*PS*) is defined as the probability that the selected DTMC produced the sequence of state-transitions in the execution E. To calculate the probability score *PS* that the DTMC-A in Figure 4.4 produced E, compute the product of the probabilities P, of the sequence of transitions in *E = {entry, S1, S2, S3, S5, S7, S8, exit}* using the values in the corresponding cells of DTMC-A:

$$PS = P(S1 \mid entry)P(S2 \mid S1)P(S3 \mid S2)P(S5 \mid S3)P(S7 \mid S5)P(S8 \mid S7)P(exit \mid S8)$$

$$PS = 1.00 * 1.00 * 0.01 * 0.50 * 0.50 * 1.00 * 1.00 = 0.0025.$$

Similarly, compute the product of the probabilities P of the sequence of transitions in *E = {entry, S1, S2, S3, S5, S7, S8, exit}* using the values in the corresponding cells of DTMC-B:

$$PS = P(S1 \mid entry)P(S2 \mid S1)P(S3 \mid S2)P(S5 \mid S3)P(S7 \mid S5)P(S8 \mid S7)P(exit \mid S8)$$

$$PS = 1.00 * 1.00 * 0.99 * 0.99 * 0.99 * 1.00 * 1.00 = 0.9703.$$

The score of 0.9703 calculated using DTMC-B is the higher of the two scores, so the classifier correctly labels E as "isosceles."

Note that probabilities calculated by multiplication can become very small. To overcome this difficulty, a standard technique for calculating probabilities using Markov

models is to convert the probabilities to their negative natural logarithms and then sum them. This transformation preserves the relative ordering of the results and thus, the integrity of the voting scheme.

4.4.4 An example procedure with loops

TriangleType serves well as an example to illustrate the basic techniques, but lacks any looping behavior. Another example procedure that does involve looping is *EvenOddCount*, which counts the number of even integers and odd integers in a list delimited by the space character and terminated with end of file (*eof*). For the purposes of this illustration, *EvenOddCount* is considered to have three external behaviors:

(1) "Even" —the input contains more even integers than odd integers

(2) "Odd" — the input contains more odd integers than even integers

(3) "Same" — the input contains equal counts of even and odd integers

Figure 4.5 shows the pseudo-code in C for *EvenOddCount* on the left and a CFG of the procedure on the right. Note that there are only two predicate statements, *S2* and *S3*, and that each is a binary decision with outcomes "true" and "false." For this example, there will be five executions of this procedure, which are listed in Table 4.1 with their inputs and their labels.

Following the technique described in Chapter 3 and shown in Figure 3.1, the procedure *EvenOddCount* is instrumented to collect branch profile data for each of the five executions specified in Table 4.1. The modeled data collected from these executions is shown in Figure 4.6 along with additional results discussed below. The top half of figure 4.6 shows five columns each representing one of the five executions *E1, E2, E3,*

```
Procedure EvenOddCount
    int i = 0;int even_sum = 0;int odd_sum = 0;
S1  read i;

S2  while (i not eof) {

S3     if ( (i mod 2) == 0)
S4        even_sum ++;
       else
S5        odd_sum ++;
       endif
S6     read i;
    endwhile
S7  printf(even_sum, odd_sum);
```

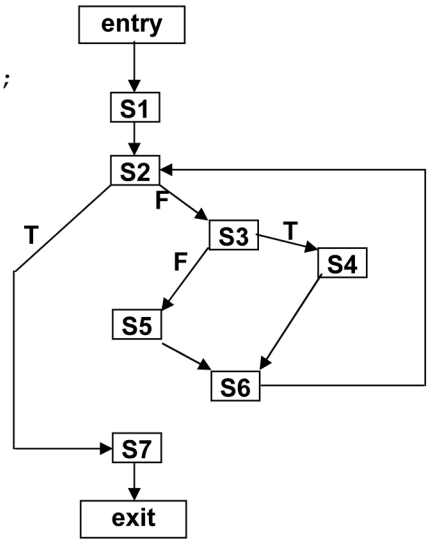


Figure 4.5. Procedure EvenOddCount.

E4, and *E5*. Below the label of each execution is the specified input and the external behavior label. For example, in the left column under the label "Execution E1" is the text "Input: 1 1 1 1 eof," and then the text "Label: Odd." Below each label in the column are three matrices. The top matrix shows the profile data collected from the execution. Note that the matrix shows only the pertinent branch transitions with the two predicate statements *S2* and *S3* labeling the rows and the four possible outcomes of these binary predicates S*3, S4, S5,* and *S7* labeling the columns. As an example, execution *E1* is shown in the first column, and the profile entry of "4" for the branch transition *S2* to *S3* represents the number of times this branch was executed in processing the specified input. Because there are four integers before the "eof" entry in the input set, this branch executed four times, and then finally the branch transition from *S2* to *S7* executed once, as represented by the "1" in the corresponding cell of the matrix. The "4" for the branch

Table 4.1. Example inputs for executions of EvenOddCount.

Execution	Inputs	Label
E1	1 1 1 1 -1	Odd
E2	1 1 1 2 -1	Odd
E3	2 2 2 2 -1	Even
E4	1 2 2 2 -1	Even
E5	1 1 2 2 -1	Same

transition *S3* to *S4* represents then number of times this branch (i.e., the integer *i* is not even, so the false branch is exercised) was taken during the execution. The second matrix in this column is the DTMC, created using the value 0.01 to replace zeroes, as described above in Section 4.4.3. The third matrix, with label "Binary" on the left side of Figure 4.6, is the binary transformation of the DTMC in preparation for using the Hamming distance as the similarity measure for clustering these executions. As described in Section 4.4.3, the threshold is 0.5 so that any entry in the DTMC that is less than 0.5 is changed to "0" and all other values are set to "1" in this binary-valued matrix.

The agglomerative hierarchical clustering algorithm begins by taking the pairwise distances, in this case the Hamming distance, between the models of each execution. As discussed in Chapter 4, this research found that the Hamming distance performed better than the absolute vector difference for the subjects studied. Because there are five executions modeled there will be ten pairs of measurements to start:

$$n*(n-1)/2 = 5*(4)/2 = 10.$$

These Hamming distances are shown in Table 4.2. These distances are simply the number of cells in which two models differ. Thus, *E1* and *E2* are identical in every cell,

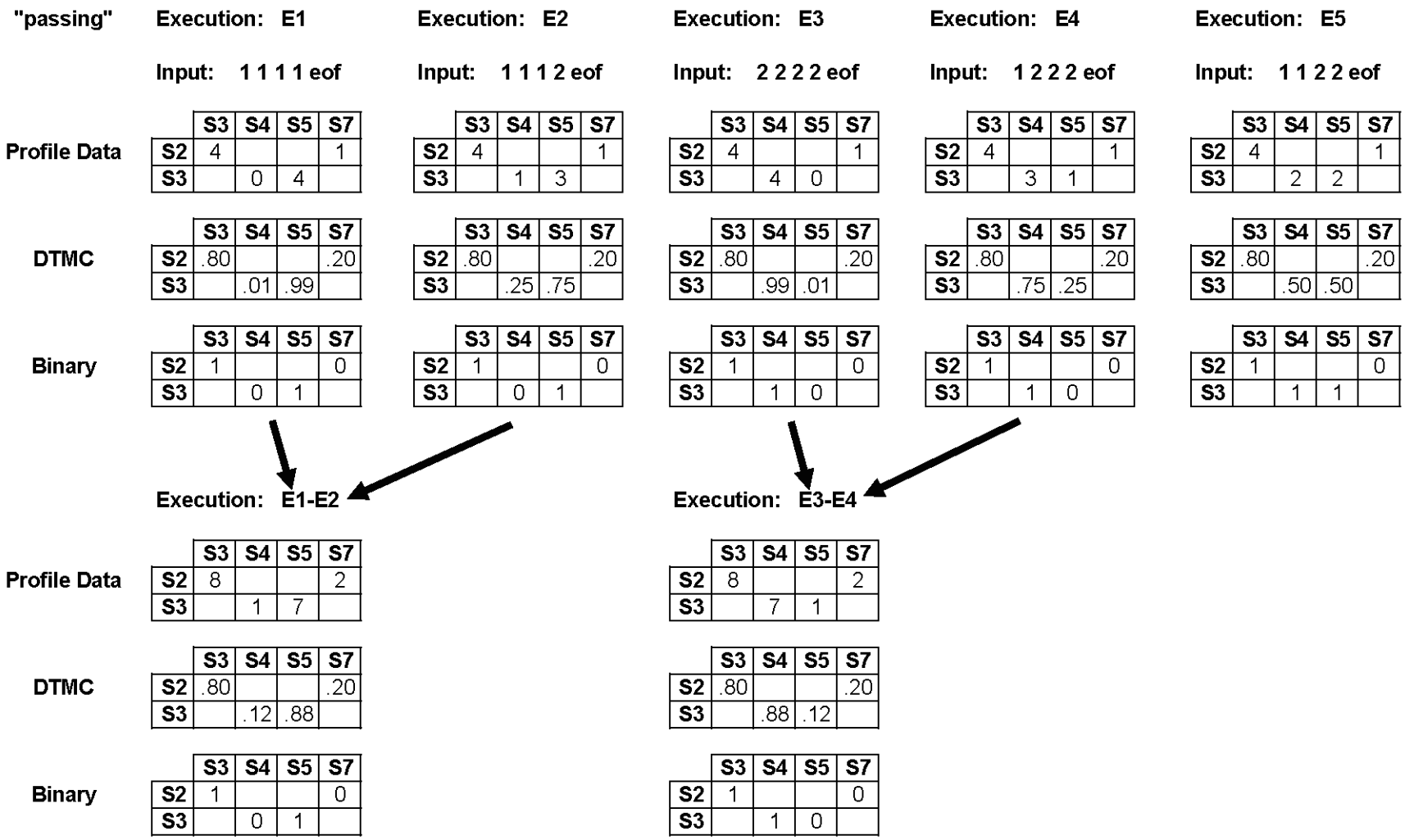


Figure 4.6. Clustering example details.

Table 4.2. Pairwise Hamming distances for example execution models.

	E1	E2	E3	E4	E5
E1	-	0	2	2	1
E2	-	-	2	2	1
E3	-	-	-	0	1
E4	-	-	-	-	0
E5	-	-	-	-	-

so the Hamming distance is zero. Likewise, *E1* and *E5* differ only in cell *S3S5,* so the Hamming distance is one. The Hamming distance is zero between *E1* and *E2*, so these two models are merged according to the method detailed in BUILDMODEL, on page 51, and the resultant merged profile data is shown in Figure 4.6 at the bottom left, in a column labeled "Execution E1-E2." Likewise, the other zero-distance pair of *E3* and *E4* is merged and shown in the center bottom of Figure 4.6. In this example, the clustering stopping criterion is three, so there are now three clusters: *E1-E2 ("odd"), E3-E4 ("even"),* and *E5 ("same").* In the empirical studies presented in Chapter 7, this research uses the heuristic described in Section 3.3 to detect automatically the stopping criterion based on clustering statistics. However, as this example is meant to illustrate the machinery of agglomeration and classification, a stopping criterion of three is used and is based in this case on domain knowledge that may in general not be available to the practitioner.

To illustrate the classification of executions of the procedure *EvenOddCount*, the three models are asked to classify the five subject executions shown in Table 4.1. Because these five models are the training set, they serve to illustrate the classification process transparently. This is done by calculating the probability that each cluster model

Table 4.3. Probability votes by each cluster to classify each execution.

	E1	E2	E3	E4	E5
E1-E2	**0.0491**	**0.0067**	0.0000	0.0001	0.0009
E3-E4	0.0000	0.0001	**0.0491**	**0.0067**	0.0009
E5	0.0051	0.0051	0.0051	0.0051	**0.0051**

produced each of the five executions. This calculation is the product of terms where each term is one of the cluster model's probabilities raised to the power of the corresponding transition profile from the execution model being classified. So, referring to the DTMC of Execution *E1-E2* in Figure 4.6, the probability that this DTMC produced the Execution *E1* is the product:

$$0.80^4 * 0.20^1 * 0.88^4 * 0.12^0 = 0.0491.$$

Similarly, each of the three models yields a probability that it produced each of the five executions, and these probabilities are shown in Table 4.3. In the table, the highlighted cells contain the highest relative probability in each column. This table shows, as expected, that the voting correctly identified each of the five executions relative to its label as "even," "odd," or "same."

The next chapter presents the data features that this research investigates with the techniques that have been presented in this chapter.

5 FEATURES AND FEATURE ENSEMBLES

The research presented in this dissertation focuses on three control-flow features and one value-flow feature of execution data that can be modeled as stochastic processes and used as the data elements for the construction of automated behavior classifiers. These features are branch profiles, branch-to-branch profiles, method-to-method profiles, and a novel feature called databin profiles. In addition to studying each of these features in isolation, the research explores how two features might be combined into an ensemble feature. This chapter details each of these features and one ensemble feature.

5.1 Control-flow features

The three control-flow features detailed below count the frequency with which program control moves from one statement to another during the program's execution. The research approach is to identify the program statements that will compose the feature's state space and that will form the prototype for the state-transition matrix and DTMC that will model the feature. Once these statements and the possible control-flow transitions among them are identified, instrumentation can be inserted into the program to count these transitions. The resulting counts are referred to as *profiles*.

5.1.1 Branch profiles

Branch profiles are defined and discussed at length in the examples and figures in Chapters 3 and 4. Branch profiles were previously studied in the development of

Software Tomography [18]. Additionally, Harrold and colleagues have shown that branch-count spectra are comparable to path count spectra for exposing fault-revealing behavior for the subjects studied, while being much less expensive to collect [51]. To accommodate both CFGs and ICFGs, the set of branches studied includes each edge from a predicate to a sink node as well as pseudo branches that denote the entry into each method. Thus, every method in a program has an "entry branch." These pseudo-branches play a bookkeeping role only and have the benefit of denoting when a method is called, even when it contains no branches itself.

5.1.2 Branch-to-branch profiles

Branch-to-branch profiles represent subpaths[11] in CFGs as well as among methods in the ICFG. The state space for this stochastic process contains the branches themselves, each of which are subpaths of length one in CFGs. A transition between two branches represents a path of at least length two in a CFG. For example, in Figure 4.5, the branch-to-branch transition between the branch with subpath (*S2, S3)* and the branch with subpath (*S3, S5)* is the subpath (*S2, S3, S5)* of length two. However, the branch-to-branch transition between the branch with subpath (*S3, S5)* and the branch with subpath (*S2, S3)*, which occurs during an iteration of the while loop, traverses edges (*S5, S6)* and edge (*S6, S2)* as intermediary subpaths. Thus, the subpath for this branch-to-branch transition is (*S3, S5, S6, S2, S3*) with length four.

The rationale for studying branch-to-branch profiles begins with the notion that a path[12] in the CFG from *Entry* to *Exit* captures the control-flow of a program execution

[11] See Definition 2.4.
[12] See Definition 2.3.

completely for a specified input. That is, for the programs studied, every time the program executes with the same inputs, the identical path executes. In a CFG that is a directed acyclic graph (i.e. with no cycles) and *n* "true" or "false" predicate statements, the number of potential paths is 2^n. The introduction of cycles into the CFG by the use of looping constructs can increase this bound substantially if each repetition of a cycle is included in the execution path. Because there is a potential for an infinite number of paths, their use for modeling execution behaviors is intractable. However, branch-to-branch transitions represent an incremental step in path length toward paths and away from the subpaths of length one that describe branches. Furthermore, the state space size for branch-to-branch transitions is equal to the number of branches.

5.1.3 Method-to-method profiles

Method-to-method, or method caller/callee, profiles have been explored by other researchers (e.g., Dickinson and Podgurski [32]) as part of a mix of features. A transition from method *M1* to method *M2* occurs when *M1* calls *M2*. If *M1* contains more than one call to *M2*, each call is considered an instance of this single transition from *M1* to *M2*. The profile of this transition is the number of times during an execution that control passes directly from *M1* to *M2*. Note that if control also passes in the opposite direction when *M2* calls *M1*, this is a separate transition. The flow of control between methods has long been recognized as an important aspect of program analysis, represented by call graphs [15].

> **Definition 5.1:** A *call graph* (*CG*) *G = (V, E)* is a directed graph of a program. Each node in the set of nodes, *V*, represents a procedure,

method, or function. Each edge in the set of directed edges, E, represents the potential flow of control between two procedures, methods, or functions. Thus, a directed edge between two nodes, u and v, is $u \rightarrow v$ and represents the potential flow of control from u to v.

In contrast with branches, the number of possible method-to-method transitions is generally between *2*n* and *n*n,* where n is the number of methods, because in the limit every method might call every other method. The rationale for studying method-to-method profiles is that they represent a higher level of transition in the flow of control of a program than do branches. In object-oriented programming languages, the emphasis is on methods, with some of the branching accomplished with dynamic dispatch. While object-oriented programs are not studied explicitly in this research, this emphasis on methods inspired the investigation of method-to-method transitions for the programs studied herein.

5.2 Value-flow feature

Despite the relationship between control-flow and both data-flow and value-flow described in Section 2.2.2, there may exist internal program behaviors that control-flow analysis cannot differentiate, but that data-based analysis may differentiate. For example, consider two versions of a program that exhibit identical control-flow for all or most of their inputs, but produce conflicting outputs. A simple illustration is a straight-line program *P*, with no conditional statements, that calculates the square of a number x. If one version of P calculates x^2 using "x * x," and another version of *P* mistakenly uses "x

+ x," both versions will exhibit identical control-flow behavior under all inputs. However, the two versions will produce the same output only when the input is either "0" or "2." In all other cases, the value-flow behaviors of the two versions of P differ. This example suggests that data-based features may capture different aspects of behaviors than control-flow features. This research investigates a novel definition of value-flows that can serve as a feature of program execution and models it as a stochastic process.

5.2.1 Stochastic models of value-flows

One goal of this research is to define and model such a feature and determine whether it usefully summarized internal program behaviors that were different from the control-flow behaviors summarized by the three features just presented. This research follows the lead of Xie and Notkin, who show that value spectra[13] can expose the root of a behavioral change in a program's execution useful for regression test selection [100]. Value-flows encapsulate properties of both a program's inputs and its execution with those inputs.

However, to model value-flows as a stochastic process requires more than value spectra. A stochastic process captures state transitions and hence in this case would capture transitions among the values taken by a variable. The challenge is to map values of variables during a program execution to a state space suitable for modeling by a stochastic process. The resulting individual stochastic models then together form a stochastic model of the internal behavior of the whole program. Figure 5.1 illustrates this larger model as a transition matrix with a diagonal composed of the individual transition

[13] See Section 2.2.2.

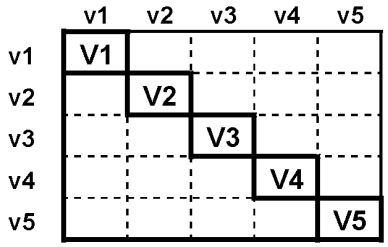


Figure 5.1. Composite transition matrix formed from individual variable matrices.

matrices for each variable.[14] This model ignores transitions among variables as a technique to reduce the number of possible transitions and hence the required storage space of the model.

There are three main concerns in modeling value-flows as state transitions of the values of variables. First, most programs use a wide range of variables and types. Tracking the values of all these variables is at best expensive and at worst intractable. Second, a single variable's range of values might differ between executions, making it difficult to specify the state space in a useful way—a state space that transcends individual executions. For example, suppose during one execution of a program the integer variable *var* takes on values in [0, 1], and during another execution this variable *var* takes on values in [103, 105]. There is no obvious mapping between the two ranges of values, other than each has a minimum and a maximum. The challenge is in identifying a state space that can represent both ranges meaningfully by normalizing the values of the variable *var* so that a single state space can apply to any execution. Third, a single variable's range of values within one execution might be so large as to make enumerating the state space potentially intractable. For example, a loop variable that

[14] See Section 5.2.2 for a description of the component models as databins.

iterates through all 8-bit integers would produce an unwieldy number of states if each integer were to be considered as a single state.

To solve the first problem requires a reduction in the overall number of considered variables by finding those variables that might best exhibit stochastic properties. After exploring the variable space of the subjects in this research, the following simplifying heuristic was developed to reduce the number of considered variables in the subject C-language programs:

1. Consider each field of a structure or class as a single variable, regardless of any instantiations of the structure or class. This is equivalent to considering each field as static in the parlance of object-oriented programming.
2. Consider all elements of an array as instances of a single variable if not considered by item 1 above.
3. Ignore constants and variables acting as constants.
4. Ignore variables local to any method.
5. Ignore pointer variables.
6. Ignore Booleans and variables that take only two values.
7. Consider only the first character of any string variable.

5.2.2 Databin transition models

To solve the second and third problems, the research developed a representation that partitions the range of values for a variable into a fixed number of bins. This technique is equal-width interval binning, an unsupervised discretization technique [38].

In *equal-width interval binning*, the width[15] of the bins is defined as the ratio of the width of the value range to the number of required bins. In addition, the research uses *global binning*, defined as using the same number of bins for every value range and which produces a mesh, as shown in Figure 5.1, containing a region for each variable's values [38]. These bins, named by this research as *databins*, are percentiles of the range for a given variable, where each percentile becomes a state in a transition matrix. The novelty of this approach is to specify a fixed bin count, regardless of the value range, so that the individual bins are the states in the model of a stochastic process. Fixing the number of databins effectively normalizes each variable's state space across individual executions of the program, solving the second problem listed above. Furthermore, specifying a fixed number of databins can limit the set of states for a variable, and this solves the third problem. For example, consider a databin count of three. In this case, the range of a variable's values will be partitioned or binned into three percentile ranges: the lower third (DB_1), the middle third (DB_2), and the upper third (DB_3). If the range of an integer variable *var* during an execution is [0, . . . , 8], then DB_1 represents [0, 1, 2]; DB_2 represents [3, 4, 5]; and DB_3 represents [6, 7, 8]. By considering individual databins as equivalence classes for the values of *var* across executions, this binning into percentiles of the range provides the required normalization. For instance, if the range of *var* during another execution were [0, . . . , 23], then the three databins would be DB_1 : [0, . . . , 7], DB_2 : [8, . . . , 15], and DB_3 : [16, . . . , 23].

For each execution of a program, the databin profiles are collected into a transition matrix and then converted to a DTMC as described in Chapter 4. For example,

[15] The width of a range is the maximum value minus the minimum value.

consider an execution of a program with variable *var* and a specification for three databins. *Var* gets the following sequence of values during an execution:

0, 1, 2, 6, 1, 4, 8, 8, 5, 5, 1.

The range is [0, . . . , 8], and therefore the three databins are DB_1: [0, 1, 2], DB_2: [3, 4, 5], and DB_3: [6, 7, 8]. The value sequence translates into a sequence of databins:

$$DB_1,\ DB_1,\ DB_1,\ DB_3,\ DB_1,\ DB_2,\ DB_3,\ DB_3,\ DB_2,\ DB_2,\ DB_1.$$

This sequence of databins is a sequence of states. The transition profiles for each state transition are obtained by traversing the sequence. The resulting transition matrix is shown in Figure 5.2 on the left and the corresponding DTMC is shown on the right.

5.3 Feature ensembles

One initial goal of this research is to understand and model individual features of program executions. A next step is to combine two features and study the properties of the combination. Machine learning research has developed *ensemble classifiers* that combine two or more classifiers that each view a set of data from different but complementary perspectives [47]. If the ensemble is beneficial, then its classification rate is better than the classification rates of its component classifiers. In this research, ensemble classifiers are composed from two different stochastic feature classifiers: branch-profile and databin-transition-profile. The empirical studies presented in Chapter 7 examine the performance of these ensembles and shows that these ensemble classifiers can outperform their component classifiers. This finding of improved classification rates for ensembles suggests that control-flow and value-flow features of program executions may capture diverse statistical views of behaviors.

Transition Matrix

databin	DB1	DB2	DB3
DB1	2	1	1
DB2	1	1	1
DB3	1	1	1

Markov Model

databin	DB1	DB2	DB3
DB1	0.50	0.25	0.25
DB2	0.33	0.33	0.33
DB3	0.33	0.33	0.33

Figure 5.2. Databin example transition matrix and Markov model.

Hansen and Salamon show that the necessary and sufficient conditions for ensemble classifiers to outperform their component classifiers are that the components each have classification error rates less than 0.5 and that they have differing classifications of the same test data [47]. Ensemble classifiers fall into two general categories: those that manipulate the data directly during training, and those that bring together existing classifiers. The first category includes bagging and boosting, which train iteratively on weighted data sets [21, 90]. The second category includes weighted voting algorithms [65]. The approach in this research is to use binary feature ensembles of classifiers: one classifier based on control-flow profiles and the other based on databin-transition profiles. The classification process is a simple weighted voting scheme [65]. In this scheme, the two component classifiers report their respective labels, including "unknown," and their confidence in these labels. The votes compare the labels reported by the two classifiers. If the two classifiers agree, the common label is used to label the execution. If they disagree, the classifier with the higher confidence wins the vote and its label is used to label the execution.

Ensembles provide an elegant mechanism to integrate the many features of data that can be collected from an executing program. For example, an ensemble might

combine a classifier derived from features profiling the program's interaction with the run-time environment and a classifier derived from one or more control-flow or data-flow features of the internal behavior of a program. The empirical studies in Chapter 7 lend confirmation to this intuition that ensemble classifiers are a useful way to capture the diverse information contained in different simultaneous profiles of an executing program.

The next chapter explores the use of the presented techniques in four categories of applications within software engineering.

6 USES IN SOFTWARE ENGINEERING

This chapter explores four categories of use for the techniques developed in this research and illustrates each of them with an example application of the techniques. Then, in Chapter 7, the empirical studies that support these example applications are presented.

6.1 Software testing

The techniques developed in this research can be applied to the automation of certain software testing tasks. This section describes one such task—automating the augmentation of test suites.

This application illustrates how a developer *Dev* can combine the techniques developed in this research for classifying behaviors to reduce the cost of extending the scope of an existing test suite (i.e., of augmenting automatically the test suite.) Other researchers have shown the potential for automatically augmenting test suites based on various behavioral characteristics (e.g., [49, 61, 74].)

Dev designs and implements a version of a program *P* for release including a test suite for testing *P* and for testing future releases of *P*. Test suite development is expensive and developers often release software that has been tested and accepted only for some core functionality [72, 81]. The goal of creating new test cases for a test suite is to test additional internal behaviors of the program. The design of a test case involves first selecting test data that will induce a new internal behavior and then evaluating the output of executing *P* with the test data. One measure of quality of the test suite is its rate

of growth as it incorporates new behaviors. As a test suite grows to include test inputs for the known or specified behaviors of a program, then it should become increasingly difficult to create additional tests that contribute additional behaviors. In this sense, *Dev* will find it difficult to add test cases for new behaviors to a high-quality test suite. However, in this scenario, *Dev*'s current test suite is lacking and *Dev* seeks an economical way to augment automatically the test suite. Even if *Dev* has an automated test data generator for *P*, *Dev* will still incur the expense of explicitly evaluating the output of each new execution.

This example application combines the technique for building classifiers presented in Chapter 3 with a use of active learning that dynamically refines the classifiers, as presented in Chapter 2. Figure 6.1 is a dataflow[16] diagram of this application. This figure is an adaptation of Figure 3.1 that shows the two-stage technique for building a behavior classifier. The detail of Stage 1 in Figure 3.2 shows how this example application will proceed. First, in Process I, *Dev* instruments *P* to collect branch profiles, for instance. Then, in Process II, *Dev* executes the instrumented program $\hat{P}$ with the tests in the current test suite. In Process III, *Dev* can label each execution because the engineers have determined the correct output for each test. The engineers' knowledge of these outputs serves the role of a behavior oracle. For example, if the engineers specify that the output for a test is *Out* and the actual output is *Out*, then the behavior label is "passing." Otherwise, the label is "failing." Thus, in Figure 6.1, Stage 1 of the technique produces the data store of training instances from the initial test plan

[16] See Definition 3.1.

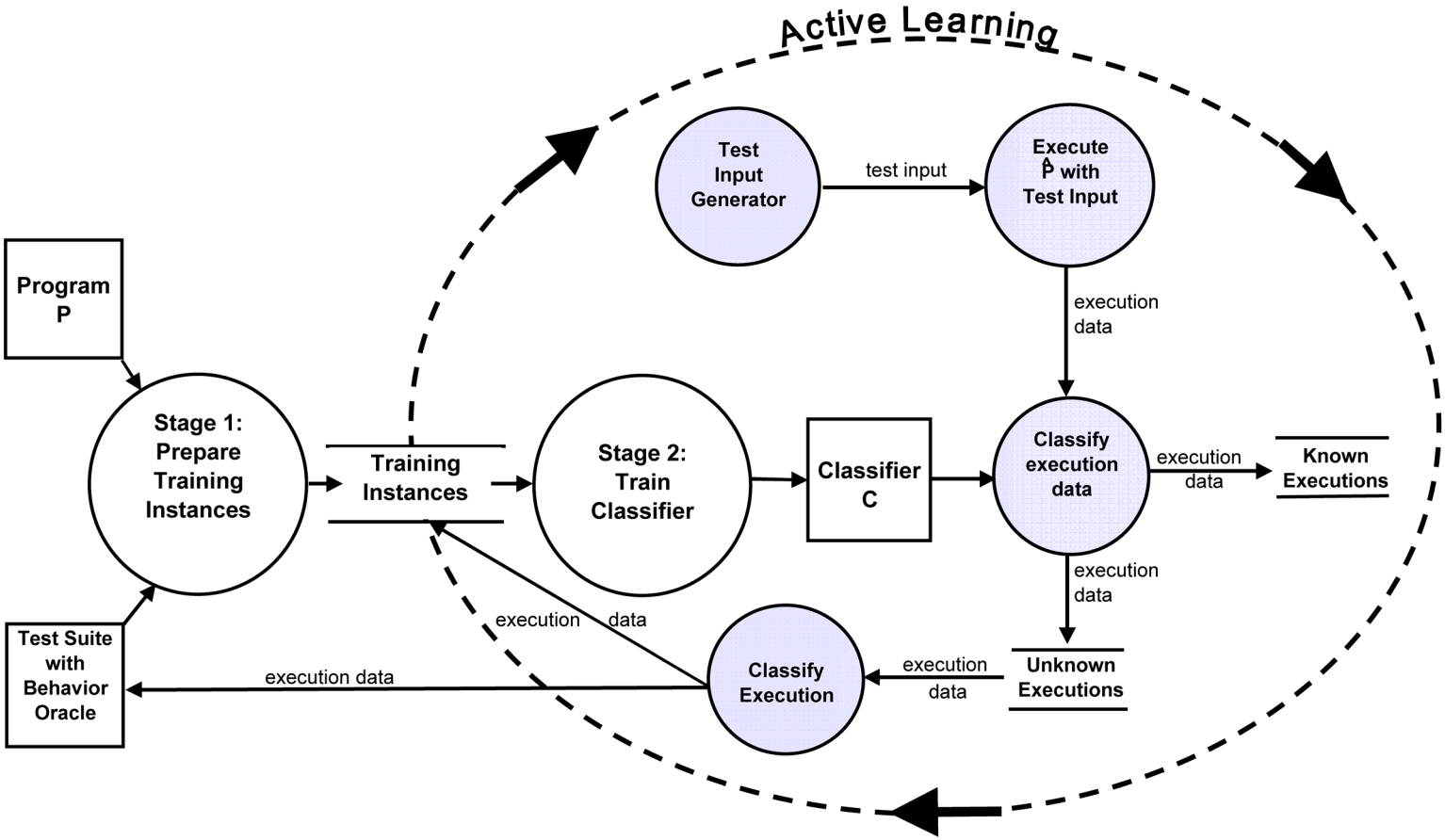


Figure 6.1. Dataflow diagram of automated test-suite augmentation.

shown as the output of Stage 1. Next in Figure 6.1, Stage 2 of the technique produces a classifier *C* for the program as detailed in Figure 3.3.

The right side of Figure 6.1 shows how this example application for *Dev* can be implemented using the classifier *C* and the active learning paradigm for refining the classifier. Figure 6.1 shows this cycle of classifier refinement within the dashed ellipse flowing in the direction of the three arrows and labeled "Active Learning" that encircles the right hand two thirds of the dataflow diagram. In this example, *Dev* does have a test input generator, represented by the shaded process circle in the top center of the dataflow diagram. *Dev* will benefit from lower costs if the classifier *C* can decide automatically whether each new test input from this generator induces a new internal behavior of the program or does not. The next step then is for the test input to flow to the process "Execute $\hat{P}$ with Test Input." This process produces *execution data* including, in this case, the branch profiles and the output of the execution. This execution data flows to the process "Classify Execution Data," which uses *C* to decide whether the data represent a *known* execution behavior or whether they represent an *unknown* execution behavior. If the behavior is recognized, that execution data is set aside into the data store labeled "Known Executions." Otherwise, the execution data is entered into the data store labeled "Unknown Executions."

If the execution resulting from the test input is known, the dataflow repeats by processing another test input if it is available. If, however, the execution was unknown, then the execution data flows to the process "Classify Execution." In this process *Dev* incurs the expense of an engineer evaluating the execution and labeling it as "passing" or "failing." From this process two dataflows occur. The first is that the execution data is

added to the existing test suite as a new exemplar, along with information for the oracle, thereby augmenting the test suite. The second dataflow is that the execution data go to the data store of training instances. At intervals, the classifier is trained on the expanded data store of training instances in the process labeled "Stage 2: Train Classifier," and the cycle repeats. By this refining process, *C* incorporates the new training instance and enriches its model to include the new behavior. Then, if a similar execution were to appear later in the cycle as a result of another generated test input, *C* would deem it as a known behavior and set it aside into the data store for known executions, thereby avoiding the cost of hand classification. The participation of classifier *C* in choosing data elements is an application of the active learning process.

Sorting test inputs into known and unknown categories is a source of the cost savings for *Dev*. Without this filtering mechanism, *Dev* would incur the cost of evaluating and labeling every execution produced by the test input generator. The costs in this example solution are directly proportional to the number of executions that must be evaluated. At the limit, active learning may require the evaluation of every new execution, thereby providing no savings over batch learning. However, in the empirical evaluations of active learning with these classifiers presented in Chapter 7, the rate at which new executions exhibited unknown behaviors decreased as the number of new test cases increased. As a second benefit of this application, *Dev* can use the rate at which the solution discovers unknown executions as a relative measure of quality describing the test suite. That is, if *Dev*'s test input generation process produces fewer and fewer new executions over time, then *Dev* might conclude that the test suite was improving in quality. *Dev* could use this rate of growth to learn a threshold at which to stop generating

new test inputs. Note that there may be additional costs related to incorrect classifications.

This example application extends to leveraging deployed software and the users of the deployed software in augmenting test suites automatically. When developers release a *beta* version of software, they seek user feedback, especially the bugs discovered while the users exercise the new software. By agreement with a sample of the customers who use *P*, *Dev* instruments their deployed copies of *P* to produce branch profiles, for example, and to upload them via the Internet to a central repository. This repository is the data store that supplies the Process "Classify Execution Data" in Figure 6.1. Then, after classification, if the execution is labeled as "unknown," the process "Classify Execution" is modified slightly. In this modification, to produce a new test case from the execution data, *Dev* must contact the users and query them about the execution and its inputs. In this way, *Dev* automates part of the reporting process with the selected customers generating test inputs. These customers are only contacted when *C* labels their execution data as "unknown." If, after a period, the selected set of customers is no longer producing many new "unknown" executions, then *Dev* can terminate the data collection, or alternatively select another sample of customers to monitor. *Dev* benefits from using this example solution to have customers produce test inputs for *P* rather than paying to generate test inputs in-house. However, *Dev* has the associated costs of collecting the data and interpreting the results. This technique was one of several proposed and outlined for the Gamma Project [73].

6.2 Failure detection in deployed software

The techniques developed in this research can be applied to automating the detection of behaviors in deployed software. This section describes one such task—detecting new behaviors and especially failures from data collected about users' executions after deployment. This example application illustrates how a developer *Dev* can use the techniques developed in this research to train a classifier for detecting automatically new behaviors in deployed software.

Dev designs and implements a version of a program *P* for release including a test suite for *P* and for future releases of *P*. There may be test cases that fail, but *Dev* releases the software anyway, which is a common practice [72, 81]. Using the technique summarized in Figure 3.1, *Dev* trains a classifier using the test suite as the training set. As part of directing the future development process, *Dev* seeks to understand whether customers are using the software in a way similar to how the test suite exercises the software. In particular, *Dev* wants to discover whether there are any new "unknown" behaviors of the program and whether they are "passing" or "failing." By agreement with selected customers, *Dev* instruments each deployed copy to produce the requisite data and transmit it to *Dev*. *Dev* uses the classifier to classify this data and to detect the existence and frequencies of "known" and "unknown" behaviors. Any "unknown" behavior becomes a candidate for an anomaly, a failure, or a new use of the program. If the classifier had trained on any known "failing" test cases, then the classifier will also detect the frequency of "known" failures. *Dev* can explore detected behaviors further by querying the customers who produce them.

6.3 Fault localization

The techniques developed in this research can be applied to the automation of debugging tools. This section describes an example application to assist fault localization. The motivation for this example application is to reduce the developer time and testing time devoted to locating the faults in a program by parallelizing the searching process. This parallelization is in contrast to the prevalent approach of sequentially locating faults.

In this application, the developer *Dev* has a software product *P* and a test suite *T* for the current version of *P*. The weekly build of *P* is run against *T,* and a number of tests fail; this subset of *T* is T_f. *Dev* uses a software tool, such as *Tarantula* [54], to process data collected from failing executions of a program *P*. The tool produces a ranking of statements based on how likely each statement is to contain the fault that caused the failure. The program may contain more than one fault, and each of these faults may or may not contribute to any one failing execution. The prevalent practice for fault localization is *sequential*—the engineers find and remove a fault, and then they re-test the program on T_f to determine if it still fails and how often. If the program still fails, they repeat the fault-removal process.

This sequential fault-localization process may not be optimal in terms of costs. For example, consider two faults, F_a and F_b, in different parts of the program that are causing some number of failures. If *Dev*'s goal is to find the faults as quickly as possible, then one optimization would be that *Dev*'s engineers could locate both faults at the same time, rather than one followed by the other. One solution to this problem is to cluster automatically the set of failing executions T_f into three groups: failures due exclusively or primarily to F_a, failures due exclusively or primarily to F_b, and failures due to both F_a

and F_b. Any one or two of these groups might be empty, depending on the effect of the specific faults. *Dev* uses the statement-ranking tool on each cluster of failures and produces a ranked list of suspicious statements for each. Then each of the resultant clusters of failing executions along with its statement ranking is given to one of *Dev*'s engineers with instructions to locate the fault. The ideal optimization here is that there are two clusters of failing executions and that each cluster produces a ranking of statements that points to a different fault. Once each fault is located and fixed, then the program is run again against *T*. If the new version of *P* does not fail, the debugging process is complete. If there are still failures, then either the faults were wrongly corrected, or there exist additional faults. In either case, this process could repeat until all the faults were removed.

When two or more faults can be located at the same time, then the process can be called *parallelized fault localization*. The principal benefit of parallelization is in reducing the overall time to produce a fault-free version of the program compared to using the sequential process. There are other cost considerations such as whether there are enough engineers available. For the purposes of this example and the empirical study supporting it in Chapter 7, the comparative measure of interest is the time to completion. To do so requires some simplifying assumptions. First, the cost of running *P* against *T* is constant for each occurrence. Second, there are more software engineers than faults available at any given time to work on debugging *P*. Without this second assumption, then in the event that there was only one engineer, the fault-localization process would occur sequentially regardless of the clustering process.

In the application of parallelized fault localization, the process depends on perfect knowledge to form the clusters. These clusters can be formed automatically, using the techniques developed in this research. Dickinson and colleagues show that clustering of executions can isolate failing executions [32, 79]. Podgurski and colleagues extend this work with an approach using multivariate visualization to aid the developer in determining whether the failing executions in a given cluster have related causes [79]. The authors find that failures can cluster because of a common cause, but provide no automated fault-localization strategy, relying instead on user interaction. In contrast, this example application seeks to cluster failing executions to isolate automatically individual faults. The empirical study in Chapter 7 supporting this application compares the costs of performing sequential versus parallelized fault localization on a set of subjects know to each have eight faults.

6.4 Software self-awareness

The software gimbal described in the Introduction is the overall motivating category of use for these techniques and inspired this research. While a gimbal remains a goal, much foundational work remains to be done.

The software gimbal scenario illustrates the potential uses of the techniques developed in this research and has been previously detailed as TripWire [19]. This scenario explores the possibility of real-time assessment of software behaviors and real-time responses to those assessments. The inspiration for this approach is the success of automatic speech recognition systems, which can assess the likelihood of a potentially unbounded set of utterances and select the most likely candidate in real-time, given an

underlying model of the conversational domain [82]. However, the analogy is strained at best, because voice recognition depends to a great degree on a large set of labeled phonemes from which utterances are constructed. There is not a set of equivalent features for executing programs.

In searching for a domain that might better yield useful tools, this research explored the idea of detecting motifs or patterns in features of execution data. For example, a program has a defined set of methods or procedures. During the execution of a program, the methods occur in sequence. If each method is assigned a unique symbol from an alphabet, then the sequence of methods produced during an execution of a sequential program can be represented as a sequence of these symbols. Similar datasets are used to encode protein sequences in computational biology, for instance. Researchers seek to identify recurrent patterns or motifs in sequences of proteins.

As a first step towards finding a way to detect behaviors in real-time, this research sought to investigate whether patterns or motifs in method call sequences might be useful. Baldi and colleagues explore the use of motifs for finding meaningful patterns in sequences of Web-page requests [8]. Cook and Wolf search for patterns (although they do not use the word *motif*) in traces of events recorded during software development [28]. The application of this technique might provide for the classification of external behaviors during an execution. Such detection would be a first step toward the real-time realization of tripwire facility for a software gimbal.

A case study of the use of motifs for detecting behaviors was conducted and the results are presented in the empirical studies in the next chapter.

7 EMPIRICAL STUDIES

To validate my thesis and support the example applications presented in Chapter 6, I developed an experimental infrastructure for experimenting with a set of subject programs using the developed techniques.

This chapter first presents in Section 7.1 the experimental infrastructure that includes existing as well as custom software. Then, in Section 7.2, the subject programs are presented. The next two sections present two sets of empirical studies. Section 7.3 presents the set of empirical studies that validate my thesis that statistical summaries of data collected during a program's execution can model and predict external behaviors of the program. Section 7.4 presents the set of empirical studies that support the example applications illustrating the categories of use described in Chapter 6. Finally, Section 7.5 discusses the threats to the validity of the results.

7.1 Infrastructure and subject programs

The two sets of empirical studies have in common the experimental infrastructure and the set of subject programs.

7.1.1 Infrastructure

As described in the Introduction, an overarching theme of this research is to understand how a software gimbal might be constructed. To that end, a prototype of the software gimbal was developed to support these studies. The gimbal is primarily

developed in the C# language, which runs in the Microsoft ".NET" virtual machine environment.[17] Nevertheless, the intended design is that the framework be independent of any particular subject program's implementation language because the framework processes data, not programming languages. This initial prototype of the gimbal is named Argo, and it is presented in detail in Appendix A.

The infrastructure supporting this research includes additional components. This research occurred as part of ongoing research within the Aristotle Research Group (ARG).[18] The ARG supplies the principal additional software component—the Aristotle Analysis System (AAS). AAS provides the static and dynamic analysis for C-language subjects. In particular, it extracts the CFG and ICFG as well as provides directed instrumentation services for a subject program. For historical reasons, AAS runs primarily on the Solaris operating system but produces test-based artifacts that can be used on any platform. The balance of the infrastructure consists of supporting software and scripts necessary for experimentation. For the empirical studies of databin profiles, the research also used the Daikon invariant detector.[19]

7.2 Subject programs

Table 7.1 lists the subject programs for these studies and shows averages for lines of code (LOC), number of methods, average cyclomatic complexity (CC) [67] per method, number of branches (except for GCC), number of versions used in these studies,

[17] The dotNet homepage with current specifications is: http://www.microsoft.com/net/ .
[18] http://www.cc.gatech.edu/aristotle/
[19] The Daikon home page and documentation can be found at: http://pag.csail.mit.edu/daikon/ .

Table 7.1. Table of subject programs.

Subject	LOC	Methods	CC	Branches	Versions	Test Cases
Space	9564	136	5	1228	15	13585
print_tokens	726	18	8	133	10	4130
replace	564	21	6	150	6	5542
schedule	412	18	4	74	5	2650
tcas	173	9	2	23	4	1608
tot_info	281	7	8	75	4	1052
flex	15297	163	13	2538	2	567
grep	15633	144	17	3478	3	809
make	27879	237	18	3827	3	1043
sed	11148	104	16	2560	4	1293
GCC	159700	2996	40	Not calc.	1	806

and number of test cases available. For each test case, the external behavior labels "passing" and "failing" are specified. The pedigree of these programs and their test case is described below.

7.2.1 Program *space*

The *space* program is an interpreter for an array definition language and was developed by the European Space Agency. Each version of *space* used in this research contains a single fault discovered during the program's development. The test suite for *space* was constructed from 10,000 test cases generated randomly by Vokolos and Frankl and then 3,585 test cases were created by researchers in ARG to guarantee that each executable edge in the program's CFG was exercised by at least 30 test cases [87, 95].

7.2.2 Siemens programs

The Siemens programs were developed by researchers at Siemens Lab: *tcas* models an aircraft collision avoidance algorithm; *schedule* is a scheduler; *tot_info*

computes statistics; *print_tokens* is a lexical analyzer; and *replace* performs pattern matching and substitution [52, 87].[20] The Siemens researchers also created test suites for each subject that guarantee that every executable statement, CFG edge, and definition-use pair was executed at least 30 times. Additionally, the researchers manufactured the single-fault versions of each program used in this research by altering between one and five lines of code to model real faults known to occur during development.

7.2.3 Programs *flex*, *grep*, *make*, and *sed*

The four programs *flex, grep, make,* and *sed* are from the public domain and were prepared as subjects by Do and colleagues [33]. The program *flex* is a lexical analyzer, *grep* filters inputs using regular expressions, *make* is a file-processing utility, and *sed* transforms data streams. Do and colleagues maintain these and additional subjects as well as those listed above in their Software-artifact Infrastructure Repository (SIR) for Experimentation repository.[21] For these four subjects, SIR provides test suites and artificially-created faults that can be inserted into the programs. SIR also provides software for generating fault-matrices for a given faulty version that compares the text outputs of the faulty version against the version without a fault.

7.2.4 Program *GCC C compiler*

The GNU Compiler Collection (GCC) C compiler version 3.1 is available under the GNU General Public License [44]. For the case of the GCC C compiler, this research

[20] There exists also a second version each of *print_tokens* and *schedule* that were not used in this research. These versions are very similar to the originals and thus they were deemed as not new programs.
[21] http://sir.unl.edu/content/sir.html

used the regression test suite that ships with version 3.2 to execute version 3.1 to create some failing executions of version 3.1.

7.3 Studies validating the thesis

The goal of this first set of empirical studies is to validate the thesis that statistical summaries of data collected during a program's execution can model and predict external behaviors of the program. As described in Chapter 5, this research focuses on the two external behaviors of executing programs labeled "passing" and "failing" and on four features of the internal behaviors of programs: branch, branch-to-branch, method-to-method, and databin transition profiles. Thus, demonstrating that the DTMC models built from these features can be processed to automatically predict the two external behaviors validates the thesis. This section presents empirical studies that use the techniques developed to extract and build models of internal program behaviors and to test how well they can detect and predict the two external behaviors.

These empirical studies of four features of a program's execution demonstrate the usefulness of DTMCs as statistical summaries of a program's execution by using them as the building blocks of automated behavior classifiers. The classifiers train on a set of DTMCs derived from program executions for which the external behavior label is known. To be considered successful, such a behavior classifier must correctly classify new data instances more than fifty per cent of the time [88]. This standard of usefulness means than the classifier performs better than random choice. At first glance, this seems to be a low standard. However, Schapire shows that it is theoretically possible in the limit to improve such a *weak classifier* until it perfectly classifies the training data [89, 90]. When the underlying assumption for classifier training is that the training data and

the testing data arise from the same distribution, Schapire argues that improvements to a classifier's ability to classify the training data should improve its ability to classify the testing data.

In the following sections, the empirical setup is described and then the empirical method and measure are described. There are four empirical studies presented next. Study 1 evaluates classifiers built from each of the three control-flow features described in Section 5.1 using batch learning. Study 2 evaluates classifiers built from these three control-flow features under active learning. Study 3 evaluates classifiers built from one value-flow feature, databin transitions (described in Section 5.2), under both batch and active learning. Finally, Study 4 evaluates ensemble classifiers, presented in Section 5.3, built from one control-flow and one value-flow feature.

7.3.1 Empirical setup

These studies prepare and use the subjects according to the dataflow diagram in Figure 3.1. To prepare the subject programs for the studies, each subject was instrumented to collect the data about the feature of interest, and then the instrumented subject was executed on its associated library of test inputs. The execution-specific data produced in this way are the "training instances" that are stored in the data store in the center of Figure 3.1.

7.3.2 Empirical method and measure

The empirical method follows the dataflow diagram in Figure 3.3 that the Argo software implements. First, the technique selects a training set of some specified size at

random from the database of training instances. The remaining training instances in the database become the testing set. Second, the technique builds the classifier using the selected training set. Building the classifier also involves specifying the similarity function *SIM*. Then the technique evaluates the classifier by measuring how well it classifies, or labels, the external behavior of each execution instance in the testing set. The classification proceeds by the voting scheme described in Chapter 4. The qualitative measure used to evaluate the resultant classifications is the *classification rate*, a ratio of correct classifications to all classifications.[22] As an example, suppose *C* scores 100 executions and correctly classifies 80. Then the classification rate = $80/100 = 0.8$. In summary, this base stage of the empirical method has five steps:

(1) select a subject version,

(2) select a data feature,

(3) select a training set and a testing set,

(4) build a classifier using the training set, and

(5) evaluate the classifier on the testing set.

A second stage of the empirical method explores classifier refinement using both batch learning and active learning. This stage repeats this first stage while steadily increasing the size of the training set used to build the classifier. Each such increase is a "training epoch." The classifier at each epoch is evaluated by measuring its classification rate.

[22] In this research, a low proportion of failures is handled by increasing the proportion of failures in the training set. Note that Podgurski and colleagues show that another approach is to average the proportions of correctly predicted failures and correctly predicted successes [79].

7.3.3 Study 1: Evaluating control-flow feature classifiers using batch learning

The goals of this study are to evaluate the predictive properties of the DTMC models when the classifiers are built using batch learning and also to track changes in the classification rate across batch-learning epochs as the number of training instances increased. Following the protocol described in the previous section, ten training epochs were created for each of the subject versions. The selection of ten epochs was based on experience with these subjects that showed that the rate of improvement in classification approached zero by epoch seven. The initial epoch used 25 training instances chosen at random from the training set. Then for each subsequent epoch, another 25 randomly chosen training instances were incorporated into the classifier. These selections were made without replacement, so that there were no repeated data elements. For each epoch, the classifier was built using the algorithm TRAINCLASSIFIER shown in Figure 4.3. This meant first segregating the training instances for each epoch into two groups by their behavior labels: "passing" or "failing." Then a classifier was trained for each of the two behavior groups. Finally, the two group classifiers were joined to be the subject's classifier for that epoch. Through experimentation, this research determined that the best similarity function of the four discussed in Section 4.4.1 was the Hamming distance between two DTMCs. This function was used as *SIM* in the inputs to the algorithm TRAINCLASSIFIER. To evaluate the classifier constructed at each epoch, it classified the testing set, and the measure recorded was the classification rate as defined above. For each version of each subject described in Table 7.1, this experiment was performed at least ten times for each of the three control-flow features. The one exception was *GCC*

for which it was only able to model method-to-method transitions because of the large size of the program and the limits of the experimental infrastructure.

The results are displayed in Table 7.2, which tabulates the parametric statistics about the mean classification rates. For example, the top row of Table 7.2 shows the results for subject *space.* Reading across the row, there is a column labeled "Epoch" and a column labeled "n," followed by three groupings of columns showing the results from using the three features: branch, branch-to-branch, and method-to-method. In the "Epoch" column, there is a row label for each of the ten epochs, labeled R1 through R10. In the column "n" there is an entry for each row showing the number of experimental runs used in the calculations. Within each grouping for an individual control-flow feature, there are four columns: "Mean," "SD," "SE," and "99% CI of Mean." The "Mean" column shows the mean classification rate for the classifier created at that epoch row from the feature. For example, at epoch "R1" for space using the branch feature, the mean classification rate for 150 runs was 0.541. The "SD" column displays the standard deviation of the observed means, calculated by:

$$SD = \left[\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right]^{1/2}.$$

where the variable x represents the members of the observed sample. The "SE" column shows the standard error or uncertainty, equal to the standard deviation of the mean. The standard error decreases with the square of the number of measurements:

$$SE = SD / \sqrt{n}.$$

Table 7.2. Results from Empirical Study 1.

	Epoch	n	BRANCH Mean	BRANCH SD	BRANCH SE	BRANCH 99% CI of Mean	BRANCH-to-BRANCH Mean	BRANCH-to-BRANCH SD	BRANCH-to-BRANCH SE	BRANCH-to-BRANCH 99% CI of Mean	METHOD-to-METHOD Mean	METHOD-to-METHOD SD	METHOD-to-METHOD SE	METHOD-to-METHOD 99% CI of Mean
space	R1	150	0.541	0.0894	0.0205	0.482 to 0.600	0.552	0.1988	0.0574	0.374 to 0.730	0.488	0.0868	0.0161	0.444 to 0.533
	R2	150	0.600	0.0608	0.0140	0.560 to 0.640	0.609	0.1881	0.0543	0.441 to 0.778	0.508	0.1040	0.0193	0.455 to 0.562
	R3	150	0.606	0.0501	0.0115	0.573 to 0.639	0.630	0.1890	0.0545	0.460 to 0.799	0.513	0.1114	0.0207	0.456 to 0.570
	R4	150	0.608	0.0397	0.0091	0.582 to 0.634	0.631	0.1956	0.0565	0.455 to 0.806	0.520	0.1033	0.0192	0.467 to 0.573
	R5	150	0.614	0.0434	0.0100	0.586 to 0.643	0.641	0.1897	0.0548	0.471 to 0.811	0.521	0.1109	0.0206	0.464 to 0.578
	R6	150	0.619	0.0393	0.0090	0.593 to 0.645	0.637	0.1930	0.0557	0.464 to 0.810	0.528	0.1078	0.0200	0.473 to 0.583
	R7	150	0.627	0.0355	0.0081	0.603 to 0.650	0.646	0.1882	0.0543	0.477 to 0.815	0.528	0.1098	0.0204	0.472 to 0.585
	R8	150	0.629	0.0274	0.0063	0.611 to 0.647	0.644	0.1844	0.0532	0.479 to 0.810	0.531	0.1033	0.0192	0.478 to 0.584
	R9	150	0.626	0.0277	0.0064	0.608 to 0.644	0.587	0.1571	0.0524	0.411 to 0.762	0.528	0.1086	0.0202	0.472 to 0.584
	R10	150	0.627	0.0223	0.0054	0.611 to 0.643	0.596	0.1521	0.0507	0.426 to 0.766	0.529	0.1115	0.0207	0.471 to 0.586
print_tokens	R1	100	0.720	0.1064	0.0201	0.665 to 0.776	0.828	0.3183	0.0750	0.611 to 1.046	0.843	0.1968	0.0410	0.727 to 0.958
	R2	100	0.754	0.0434	0.0082	0.732 to 0.777	0.839	0.2956	0.0697	0.637 to 1.041	0.847	0.1861	0.0388	0.737 to 0.956
	R3	100	0.761	0.0354	0.0067	0.742 to 0.779	0.842	0.2894	0.0682	0.644 to 1.040	0.857	0.1817	0.0387	0.747 to 0.966
	R4	100	0.770	0.0312	0.0059	0.754 to 0.786	0.863	0.2498	0.0589	0.692 to 1.034	0.858	0.1767	0.0377	0.751 to 0.965
	R5	100	0.769	0.0313	0.0059	0.753 to 0.786	0.880	0.2228	0.0525	0.728 to 1.032	0.846	0.1737	0.0370	0.741 to 0.951
	R6	100	0.770	0.0300	0.0057	0.755 to 0.786	0.883	0.2109	0.0497	0.739 to 1.027	0.847	0.1722	0.0367	0.743 to 0.951
	R7	100	0.771	0.0328	0.0062	0.754 to 0.788	0.858	0.2244	0.0529	0.705 to 1.011	0.844	0.1769	0.0377	0.737 to 0.951
	R8	100	0.771	0.0313	0.0059	0.754 to 0.787	0.855	0.2278	0.0537	0.700 to 1.011	0.854	0.1546	0.0330	0.761 to 0.947
	R9	100	0.768	0.0318	0.0060	0.752 to 0.785	0.865	0.2126	0.0501	0.720 to 1.010	0.851	0.1564	0.0333	0.757 to 0.945
	R10	100	0.764	0.0394	0.0075	0.743 to 0.785	0.867	0.2119	0.0500	0.722 to 1.011	0.844	0.1546	0.0330	0.751 to 0.938
replace	R1	60	0.583	0.1223	0.0387	0.458 to 0.709	0.810	0.2293	0.0691	0.591 to 1.029	0.800	0.2374	0.0716	0.573 to 1.027
	R2	60	0.618	0.1185	0.0375	0.497 to 0.740	0.836	0.1856	0.0560	0.658 to 1.013	0.824	0.1682	0.0507	0.663 to 0.984
	R3	60	0.628	0.0971	0.0307	0.529 to 0.728	0.847	0.1722	0.0519	0.682 to 1.011	0.806	0.1891	0.0570	0.625 to 0.986
	R4	60	0.643	0.0874	0.0276	0.553 to 0.733	0.847	0.1700	0.0512	0.685 to 1.010	0.792	0.1793	0.0541	0.621 to 0.964
	R5	60	0.660	0.0722	0.0228	0.585 to 0.734	0.851	0.1661	0.0501	0.692 to 1.009	0.813	0.1720	0.0519	0.648 to 0.977
	R6	60	0.667	0.0671	0.0212	0.598 to 0.735	0.841	0.1627	0.0491	0.685 to 0.996	0.817	0.1693	0.0510	0.655 to 0.979
	R7	60	0.667	0.0668	0.0211	0.598 to 0.736	0.832	0.1756	0.0530	0.664 to 1.000	0.803	0.1596	0.0481	0.651 to 0.956
	R8	60	0.666	0.0649	0.0205	0.600 to 0.733	0.838	0.1645	0.0496	0.681 to 0.995	0.786	0.1882	0.0567	0.606 to 0.966
	R9	60	0.664	0.0624	0.0197	0.599 to 0.728	0.822	0.1822	0.0549	0.648 to 0.996	0.752	0.1888	0.0569	0.572 to 0.933
	R10	60	0.657	0.0668	0.0211	0.588 to 0.726	0.820	0.1861	0.0561	0.643 to 0.998	0.805	0.2048	0.0617	0.610 to 1.001

Table 7.2. (Continued).

	Epoch	n	BRANCH Mean	BRANCH SD	BRANCH SE	BRANCH 99% CI of Mean	BRANCH-to-BRANCH Mean	BRANCH-to-BRANCH SD	BRANCH-to-BRANCH SE	BRANCH-to-BRANCH 99% CI of Mean	METHOD-to-METHOD Mean	METHOD-to-METHOD SD	METHOD-to-METHOD SE	METHOD-to-METHOD 99% CI of Mean
schedule	R1	50	0.570	0.1142	0.0167	0.525 to 0.615	0.762	0.2268	0.0655	0.559 to 0.965	0.648	0.1483	0.0303	0.563 to 0.733
	R2	50	0.534	0.1268	0.0185	0.485 to 0.584	0.664	0.2858	0.0825	0.408 to 0.921	0.525	0.1951	0.0398	0.414 to 0.637
	R3	50	0.520	0.1169	0.0170	0.474 to 0.566	0.659	0.2832	0.0817	0.405 to 0.913	0.510	0.2137	0.0436	0.387 to 0.632
	R4	50	0.496	0.1463	0.0213	0.438 to 0.553	0.625	0.2691	0.0777	0.384 to 0.866	0.445	0.2163	0.0442	0.321 to 0.569
	R5	50	0.496	0.1427	0.0208	0.440 to 0.552	0.609	0.2473	0.0714	0.387 to 0.831	0.463	0.1946	0.0397	0.351 to 0.574
	R6	50	0.487	0.1406	0.0205	0.432 to 0.542	0.607	0.2469	0.0713	0.385 to 0.828	0.490	0.1824	0.0372	0.385 to 0.594
	R7	50	0.483	0.1474	0.0215	0.425 to 0.541	0.615	0.2559	0.0739	0.386 to 0.845	0.449	0.1913	0.0391	0.339 to 0.558
	R8	50	0.494	0.1413	0.0206	0.439 to 0.550	0.590	0.2480	0.0716	0.368 to 0.812	0.477	0.1763	0.0360	0.376 to 0.578
	R9	50	0.471	0.1573	0.0229	0.410 to 0.533	0.589	0.2475	0.0714	0.367 to 0.811	0.425	0.1738	0.0355	0.326 to 0.525
	R10	50	0.470	0.1578	0.0230	0.408 to 0.532	0.609	0.2649	0.0765	0.371 to 0.846	0.460	0.1954	0.0399	0.348 to 0.572
tcas	R1	40	0.485	0.0748	0.0306	0.362 to 0.608	0.438	0.0107	0.0048	0.416 to 0.460	0.448	0.0016	0.0005	0.447 to 0.450
	R2	40	0.515	0.0345	0.0141	0.458 to 0.571	0.442	0.0106	0.0048	0.420 to 0.464	0.448	0.0023	0.0007	0.445 to 0.450
	R3	40	0.529	0.0080	0.0033	0.516 to 0.542	0.444	0.0093	0.0041	0.425 to 0.463	0.448	0.0027	0.0008	0.445 to 0.451
	R4	40	0.529	0.0082	0.0033	0.516 to 0.543	0.444	0.0097	0.0043	0.424 to 0.464	0.448	0.0033	0.0010	0.445 to 0.452
	R5	40	0.532	0.0038	0.0016	0.526 to 0.539	0.447	0.0037	0.0016	0.439 to 0.454	0.449	0.0033	0.0010	0.446 to 0.452
	R6	40	0.532	0.0055	0.0022	0.523 to 0.541	0.447	0.0041	0.0018	0.438 to 0.455	0.449	0.0028	0.0009	0.446 to 0.452
	R7	40	0.533	0.0067	0.0027	0.522 to 0.544	0.447	0.0042	0.0019	0.438 to 0.455	0.449	0.0036	0.0011	0.445 to 0.452
	R8	40	0.533	0.0065	0.0027	0.522 to 0.544	0.448	0.0033	0.0015	0.441 to 0.455	0.449	0.0027	0.0009	0.446 to 0.452
	R9	40	0.533	0.0072	0.0029	0.521 to 0.545	0.448	0.0034	0.0015	0.441 to 0.455	0.449	0.0031	0.0010	0.446 to 0.452
	R10	40	0.533	0.0070	0.0028	0.522 to 0.545	0.449	0.0028	0.0013	0.443 to 0.455	0.450	0.0037	0.0012	0.446 to 0.453
tot_info	R1	40	0.742	0.1997	0.0534	0.582 to 0.903	0.669	0.2039	0.0615	0.474 to 0.864	0.772	0.1099	0.0331	0.667 to 0.878
	R2	40	0.630	0.1355	0.0362	0.521 to 0.740	0.670	0.1713	0.0517	0.506 to 0.834	0.693	0.1614	0.0487	0.539 to 0.847
	R3	40	0.604	0.1030	0.0275	0.521 to 0.687	0.656	0.1301	0.0392	0.532 to 0.780	0.645	0.1403	0.0423	0.510 to 0.779
	R4	40	0.579	0.0660	0.0177	0.526 to 0.633	0.626	0.1458	0.0440	0.487 to 0.766	0.706	0.1235	0.0372	0.588 to 0.824
	R5	40	0.600	0.0878	0.0235	0.530 to 0.671	0.618	0.1477	0.0445	0.477 to 0.759	0.667	0.1296	0.0391	0.543 to 0.790
	R6	40	0.582	0.0768	0.0205	0.520 to 0.644	0.615	0.1096	0.0330	0.510 to 0.719	0.761	0.0951	0.0287	0.670 to 0.852
	R7	40	0.573	0.0749	0.0200	0.513 to 0.633	0.586	0.1404	0.0423	0.452 to 0.720	0.778	0.0979	0.0295	0.684 to 0.871
	R8	40	0.573	0.0740	0.0198	0.513 to 0.632	0.607	0.1258	0.0379	0.487 to 0.728	0.734	0.0969	0.0292	0.641 to 0.826
	R9	40	0.574	0.0508	0.0136	0.533 to 0.615	0.623	0.1126	0.0339	0.516 to 0.731	0.767	0.0906	0.0273	0.680 to 0.853
	R10	40	0.565	0.0473	0.0126	0.527 to 0.603	0.624	0.1192	0.0359	0.510 to 0.738	0.789	0.0580	0.0175	0.734 to 0.845

Table 7.2. (Continued).

	Epoch	n	BRANCH Mean	BRANCH SD	BRANCH SE	BRANCH 99% CI of Mean	BRANCH-to-BRANCH Mean	BRANCH-to-BRANCH SD	BRANCH-to-BRANCH SE	BRANCH-to-BRANCH 99% CI of Mean	METHOD-to-METHOD Mean	METHOD-to-METHOD SD	METHOD-to-METHOD SE	METHOD-to-METHOD 99% CI of Mean
flex	R1	20	0.575	0.1042	0.0368	0.446 to 0.704	0.762	0.2268	0.0655	0.559 to 0.965	0.526	0.0858	0.0384	0.349 to 0.703
	R2	20	0.640	0.1099	0.0389	0.504 to 0.776	0.664	0.2858	0.0825	0.408 to 0.921	0.499	0.0975	0.0436	0.298 to 0.700
	R3	20	0.660	0.1037	0.0367	0.531 to 0.788	0.659	0.2832	0.0817	0.405 to 0.913	0.539	0.0944	0.0422	0.344 to 0.733
	R4	20	0.694	0.0895	0.0316	0.583 to 0.804	0.625	0.2691	0.0777	0.384 to 0.866	0.501	0.0797	0.0356	0.337 to 0.665
	R5	20	0.665	0.0706	0.0250	0.578 to 0.753	0.609	0.2473	0.0714	0.387 to 0.831	0.510	0.0586	0.0262	0.390 to 0.631
	R6	20	0.682	0.0588	0.0208	0.609 to 0.755	0.607	0.2469	0.0713	0.385 to 0.828	0.495	0.0950	0.0425	0.299 to 0.691
	R7	20	0.686	0.0527	0.0186	0.621 to 0.752	0.615	0.2559	0.0739	0.386 to 0.845	0.469	0.0812	0.0363	0.302 to 0.637
	R8	20	0.677	0.0451	0.0160	0.621 to 0.733	0.590	0.2480	0.0716	0.368 to 0.812	0.485	0.0852	0.0381	0.310 to 0.661
	R9	20	0.681	0.0351	0.0124	0.637 to 0.724	0.589	0.2475	0.0714	0.367 to 0.811	0.536	0.0838	0.0375	0.363 to 0.708
	R10	20		no data				no data				no data		
grep	R1	30	0.584	0.0610	0.0169	0.532 to 0.635	0.901	0.2814	0.0995	0.552 to 1.249	0.752	0.2052	0.0513	0.601 to 0.903
	R2	30	0.606	0.0437	0.0121	0.569 to 0.643	0.909	0.2576	0.0911	0.590 to 1.228	0.824	0.2039	0.0510	0.674 to 0.974
	R3	30	0.621	0.0337	0.0093	0.593 to 0.650	0.911	0.2510	0.0887	0.601 to 1.222	0.905	0.0395	0.0099	0.876 to 0.934
	R4	30	0.618	0.0409	0.0113	0.583 to 0.653	0.913	0.2448	0.0866	0.611 to 1.216	0.907	0.0866	0.0216	0.843 to 0.971
	R5	30	0.622	0.0358	0.0099	0.591 to 0.652	0.914	0.2440	0.0863	0.612 to 1.216	0.926	0.0235	0.0059	0.909 to 0.944
	R6	30	0.623	0.0367	0.0102	0.591 to 0.654	0.914	0.2441	0.0863	0.612 to 1.216	0.935	0.0142	0.0036	0.925 to 0.946
	R7	30	0.626	0.0450	0.0125	0.588 to 0.664	0.915	0.2415	0.0854	0.616 to 1.213	0.938	0.0139	0.0035	0.928 to 0.948
	R8	30	0.632	0.0450	0.0125	0.594 to 0.670	0.915	0.2376	0.0840	0.621 to 1.209	0.938	0.0164	0.0041	0.926 to 0.950
	R9	30	0.638	0.0395	0.0110	0.605 to 0.671	0.914	0.2366	0.0836	0.622 to 1.207	0.943	0.0130	0.0033	0.933 to 0.952
	R10	30	0.645	0.0367	0.0102	0.614 to 0.676	0.912	0.2371	0.0838	0.619 to 1.206	0.945	0.0127	0.0032	0.935 to 0.954
make	R1	30	0.190	0.1379	0.0488	0.019 to 0.361	0.235	0.0999	0.0577	-0.338 to 0.807	0.263	0.0800	0.0283	0.163 to 0.362
	R2	30	0.149	0.0616	0.0218	0.073 to 0.225	0.368	0.2096	0.1210	-0.833 to 1.569	0.300	0.0463	0.0164	0.243 to 0.357
	R3	30	0.141	0.0274	0.0097	0.107 to 0.175	0.219	0.0937	0.0541	-0.318 to 0.755	0.315	0.1012	0.0358	0.189 to 0.440
	R4	30	0.151	0.0091	0.0032	0.140 to 0.163	0.160	0.0475	0.0274	-0.112 to 0.432	0.295	0.0542	0.0192	0.228 to 0.362
	R5	30	0.153	0.0067	0.0024	0.145 to 0.162	0.142	0.0383	0.0221	-0.077 to 0.361	0.304	0.0498	0.0176	0.242 to 0.365
	R6	30	0.153	0.0066	0.0023	0.145 to 0.161	0.129	0.0140	0.0081	0.049 to 0.209	0.291	0.0482	0.0171	0.231 to 0.350
	R7	30	0.153	0.0069	0.0028	0.142 to 0.164	0.126	0.0120	0.0070	0.057 to 0.195	0.297	0.0474	0.0193	0.219 to 0.375
	R8	30	0.156	0.0098	0.0040	0.140 to 0.172	0.123	0.0111	0.0064	0.059 to 0.187	0.286	0.0506	0.0207	0.202 to 0.369
	R9	30	0.156	0.0113	0.0046	0.138 to 0.175	0.124	0.0083	0.0048	0.077 to 0.171	0.303	0.0486	0.0198	0.223 to 0.383
	R10	30	0.158	0.0082	0.0033	0.145 to 0.171	0.123	0.0099	0.0057	0.066 to 0.180	0.296	0.0471	0.0192	0.218 to 0.373

Table 7.2. (Continued).

	Epoch	n	BRANCH Mean	BRANCH SD	BRANCH SE	BRANCH 99% CI of Mean	BRANCH-to-BRANCH Mean	BRANCH-to-BRANCH SD	BRANCH-to-BRANCH SE	BRANCH-to-BRANCH 99% CI of Mean	METHOD-to-METHOD Mean	METHOD-to-METHOD SD	METHOD-to-METHOD SE	METHOD-to-METHOD 99% CI of Mean
sed	R1	40	0.555	0.2081	0.1040	-0.053 to 1.162	0.349	0.0262	0.0131	0.272 to 0.425	0.401	0.0419	0.0187	0.315 to 0.487
	R2	40	0.643	0.1535	0.0768	0.194 to 1.091	0.347	0.0269	0.0134	0.269 to 0.426	0.497	0.0534	0.0239	0.387 to 0.607
	R3	40	0.680	0.1299	0.0650	0.300 to 1.059	0.360	0.0030	0.0015	0.351 to 0.368	0.497	0.0393	0.0176	0.416 to 0.578
	R4	40	0.671	0.1070	0.0535	0.359 to 0.983	0.334	0.0489	0.0245	0.191 to 0.477	0.506	0.0439	0.0196	0.415 to 0.596
	R5	40	0.642	0.1414	0.0707	0.229 to 1.055	0.299	0.0714	0.0357	0.091 to 0.507	0.516	0.0355	0.0159	0.442 to 0.589
	R6	40	0.632	0.1261	0.0631	0.264 to 1.000	0.299	0.0720	0.0360	0.089 to 0.510	0.522	0.0453	0.0203	0.429 to 0.615
	R7	40	0.642	0.1442	0.0721	0.221 to 1.063	0.325	0.0495	0.0247	0.180 to 0.469	0.517	0.0314	0.0140	0.453 to 0.582
	R8	40	0.635	0.1625	0.0813	0.161 to 1.110	0.334	0.0530	0.0265	0.179 to 0.489	0.522	0.0285	0.0127	0.463 to 0.580
	R9	40	0.585	0.1537	0.0768	0.137 to 1.034	0.324	0.0508	0.0254	0.176 to 0.473	0.523	0.0189	0.0085	0.484 to 0.562
	R10	40	0.572	0.0707	0.0354	0.365 to 0.779	0.324	0.0488	0.0244	0.182 to 0.467	0.511	0.0126	0.0056	0.485 to 0.537
GCC C compiler	R1	10									0.129	0.0389	0.0138	0.081 to 0.177
	R2	10									0.278	0.0658	0.0233	0.196 to 0.359
	R3	10									0.401	0.0812	0.0287	0.301 to 0.502
	R4	10									0.485	0.0513	0.0181	0.421 to 0.548
	R5	10									0.536	0.0574	0.0203	0.465 to 0.607
	R6	10									0.562	0.0459	0.0162	0.505 to 0.619
	R7	10									0.596	0.0216	0.0076	0.569 to 0.622
	R8	10									0.618	0.0113	0.0040	0.604 to 0.632
	R9	10									0.612	0.0471	0.0166	0.553 to 0.670
	R10	10									0.618	0.0119	0.0050	0.603 to 0.633

The "99% CI of Mean" column shows the ninety-nine percent confidence interval for the mean—99% of the sample occurs within this range.

For the branch profile feature, two of the ten subjects— *make* and *schedule*—do not produce classifiers trained with batch learning that are able to perform above random choice. The remaining eight subjects generally perform at a classification rate of 0.6 or higher. However, for the branch-to-branch feature, *make, sed,* and *tcas* perform below 0.5, while *schedule* performs at about 0.6. For the method-to-method feature, *schedule, make,* and *tcas* perform at or below 0.5. Across all subjects and all three features, the classifiers trained with batch learning generally do not improve with the larger training sets associated with each advancing epoch. This research sought to find a technique to refine the classifiers at each epoch so that their classification rate might improve. The technique chosen is active learning, as described in Section 2.3.1. The next study explores the application of active learning to the same subjects and the same features examined in this study.

7.3.4 Study 2: Evaluating control-flow feature classifiers using active learning

One goal of the second study was to evaluate the predictive properties of the DTMC models when active learning was used to build the classifiers. Additional related goals were to track changes in the classification rate across active-learning epochs as the number of training instances increased and to compare the results of active learning with those of batch learning in Study 1. As in Study 1, for each of the subject versions, ten training epochs were generated for each of the subject versions. The initial epoch used a training set size of 25 and each new epoch increased this size by an additional 25 training

instances. To aid in comparing results with Study 1, the initial training set from the first epoch (training set size of 25) in Study 1 was used in this study for the classifier at its first epoch.

The active-learning paradigm introduces another decision into the classification process. In active learning, the classifier must be able to recognize whether a new data item is known or unknown, as described in the example scenarios for test-suite augmentation in Section 6.2. In other words, the classifier must also determine a level of confidence in its decision to label an execution "passing" or "failing." For example, if the DTMC of a new execution's data is very dissimilar from all of the DTMC training instances incorporated into the classifier, the classifier should report that this new execution is "unknown" because the probabilities that it is either "passing" or "failing" are very low relative to some threshold. It is this decision to label a data element as "unknown" that feeds the active learning process, for these "unknown" elements are then chosen for inclusion into the training set for the classifier, after first being labeled by hand as "passing" or "failing." Thus, any data elements that the classifier labels "passing" or "failing" are considered as "known," and the active learning paradigm sets these aside as not providing new information for training purposes.

The establishment of this threshold value is a heuristic process. This research arrived at the following heuristic for the subject programs in this study:

> The first step at each training epoch is to train the classifier on the selected training set. Then, to establish the threshold, the classifier classifies its own training set. The threshold used in these studies is the mean value of the probability scores that the classifier produces when

labeling the training set. The classifier will label any test data element as "unknown" if its probability score is below this heuristically determined threshold.

With active learning, at each epoch the choice of additional training instances from the training set was made by using the current classifier to classify randomly chosen instances from the training set until the classifier found the requisite number, in this case 25, of additional executions with "unknown" behaviors. (For this experiment, the stored fault matrix for the subject provided the correct behavior label for each of these "unknown" executions.) The classifier then incorporated these new instances into its training set for the next epoch. This process is known as *classifier refinement*. If the classifier finds no additional "unknown" training instances in the training set, the classifier is characterized as stable for this data set and does not undergo further refinement in the subsequent epochs of this experimental run.

Figure 7.1 summarizes as graphs the results for all subjects and all classifiers evaluated for both active learning and batch learning. In this figure (printed on four pages), each row has three graphs for each subject, with the exception of *GCC*. Reading left to right for each subject, the graphs depict comparatively the results of both Study 1 and Study 2 for the three features: branch, branch-to-branch, and method-to-method. Within each graph, the batch-learning results of Study 1 are shown with solid triangles for the mean classification rate at each epoch and are connected with a dashed line.

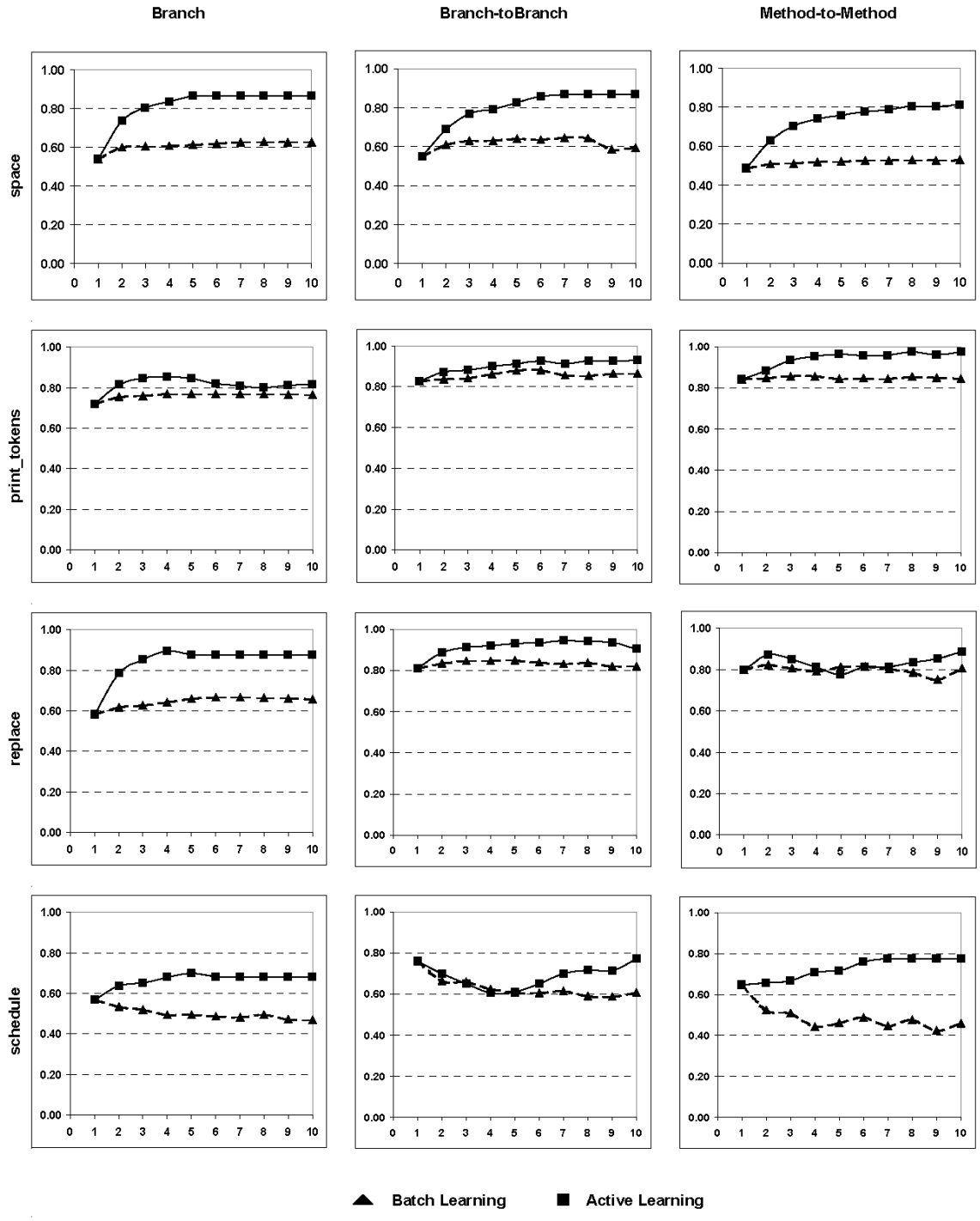


Figure 7.1. Comparison of batch and active learning for three features.

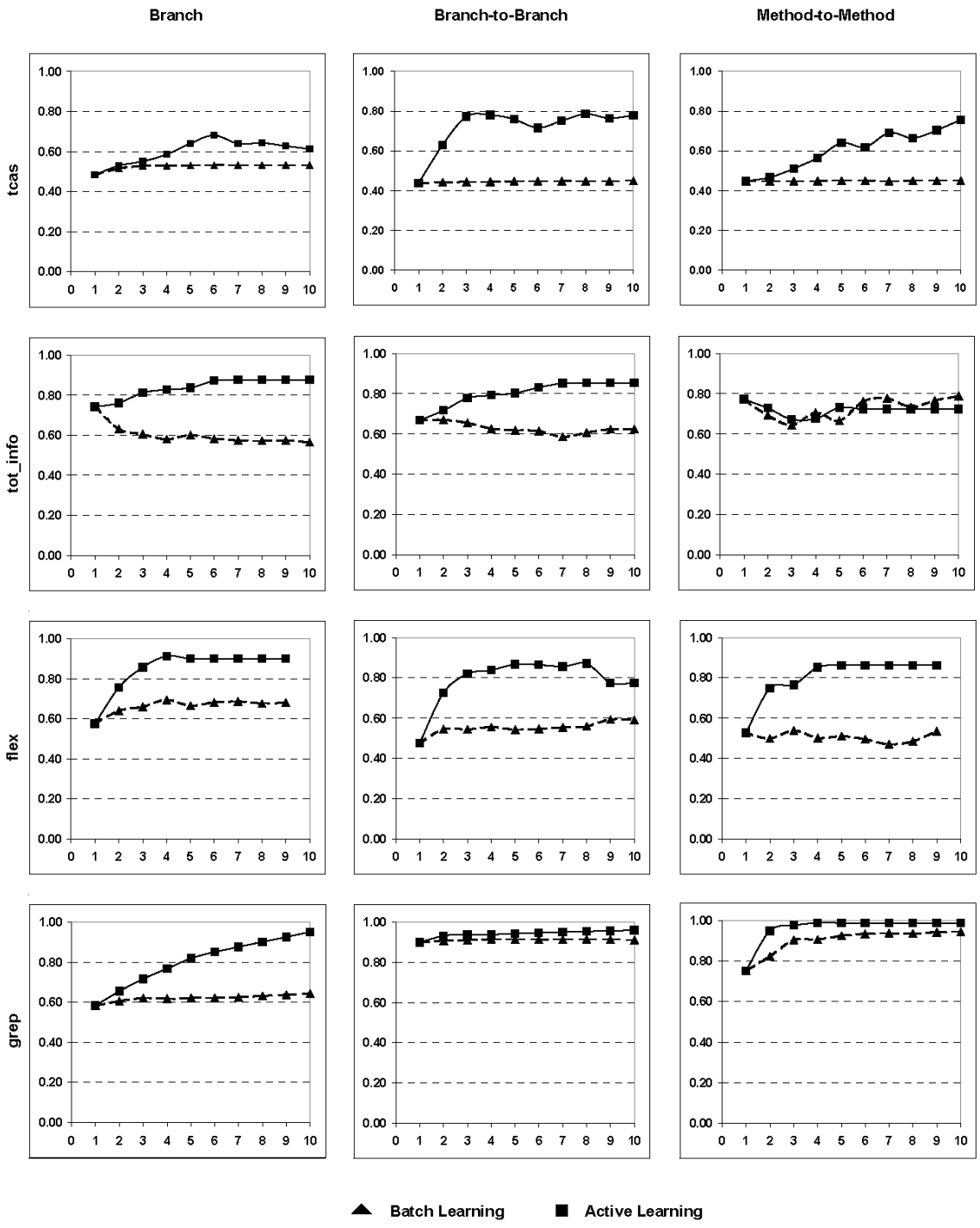


Figure 7.1 (Continued).

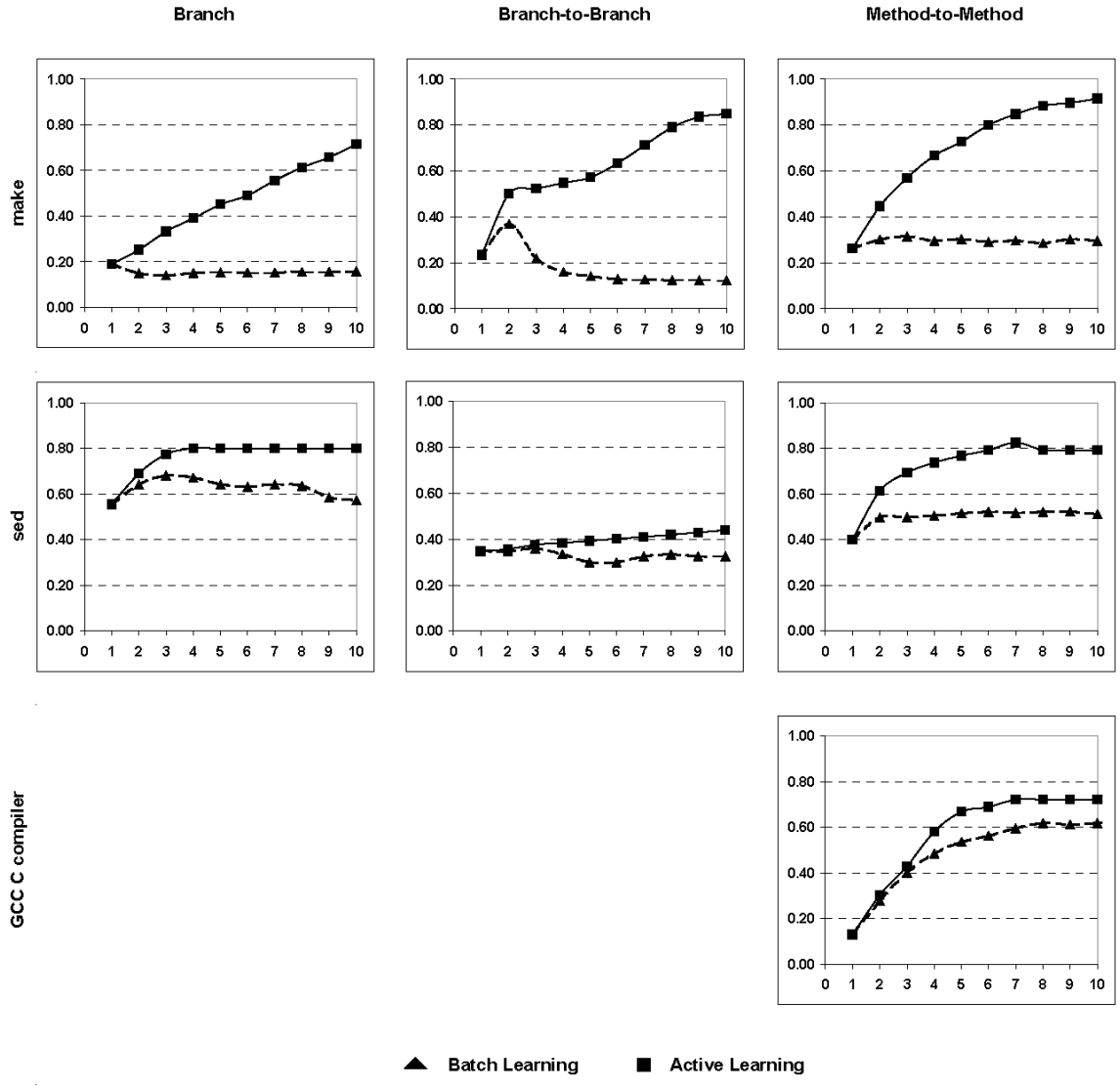


Figure 7.1 (Continued).

Similarly, the active-learning results of this Study 2 are shown with the solid squares and solid lines.

As an example, the top row of graphs in Figure 7.1 is for the subject *space* and the left-most graph in the row for *space* shows the results of both studies for classifiers built from DTMCs of branch profiles. Within this graph, the vertical axis shows the mean classification rate as between 0.00 and 1.00, and the horizontal axis shows the ten training epochs by ordinal number from 1 through 10. In the example graph for *space*, the plot of mean classification rates for batch learning uses the data shown in Table 7.2 and shows graphically that this rate remains approximately 0.6 for all epochs. In contrast, in this same graph, the plot of the mean classification rates for active learning steadily improves between epoch 1 and epoch 10 to a mean value of approximately 0.9.

These results for active learning show that in general, for each of these subjects, there is a significant improvement in the rate of classification between epoch 2 and epoch 10 relative to batch learning for at least one of the features studied for that subject. There exists variance between subjects and, within subjects, between features, in the predictive power of these classifiers, as measured by their classification rates. For example, the branch and branch-to-branch features for *space* produce equivalent classifiers at epoch 10 with classification rates of approximately 0.9, while method-to-method yields a classification rate of 0.8 at epoch 10. As another example, the method-to-method feature produces the best result at 0.98 for subject *print_tokens*.

There are a few abnormal results as well. The active-learning performances of *replace* under method-to-method, *schedule* under branch-to-branch, and *tot_info* under method-to-method are three such cases. In each, the classification rate under active

learning deteriorates for several epochs before increasing. An explanation is that this may reveal a property of the available test cases. The *tot_info* results show no appreciable difference between batch learning and active learning. The likely reason is that *tot_info* has only seven methods compared to 75 branches (see Table 7.1.)

For the ten subjects studied with all three features, the results are split evenly as to whether branch profiles or method-to-method profiles are superior under active-learning. For a developer, this choice can be made based on empirical studies and cost. The respective costs for the two features are dependent on the subject program. While the number of methods and, hence, the number of states in the DTMC will be generally less than the number of predicates in all but the simplest programs, the number of transitions between methods may be greater than the number of branches. In the case of subject *GCC*, the number of methods is 2,996 and this pushed the limits of the Argo infrastructure. To minimize the space requirements for the DTMCs requires the use of sparse matrices and therefore, the number of possible transitions in a DTMC directly affects the storage costs for a DTMC. For example, *space* has 136 methods and 1,040 method-to-method transitions compared to 1,226 branches. The models for *space* are of the same magnitude. In contrast, *grep* has 144 methods and 646 method-to-method transitions compared to 3,478 branches. The models for *grep* differ by an order of magnitude. For *grep*, a developer would choose the method-to-method profile because its models are smaller than for branch profiles and because, under active learning, these models produce a better classifier than the branch-profile models, as shown in Figure 6.4. A second cost consideration is that of data collection. Both branch profiles and method-to-method profiles can be collected by instrumentation of the program. Method-to-

method profiles can also be collected by monitoring the program's stack. A developer faced with these decisions would likely perform comparative empirical studies to examine these costs and the predictive power of the three features for a software product for which behavior detection was important.

7.3.5 Study 3: Evaluating value-flow classifiers with batch and active learning

A goal of this study was to evaluate the performance of databin-transition-based classifiers under both batch and active learning. Thus, this study is similar to Study 2 above, except that it focuses on this one feature of databins, detailed in Section 5.2.2. The initial subjects for these studies were the ten C programs (excluding *GCC*) shown in table 7.1. However, *tcas* and *tot_info* are both very simple programs that depend almost entirely on a "main" method and they did not yield any variable suitable for databins. Also, *grep*, *make,* and *GCC* were too large for the databin extraction infrastructure described below and hence were not studied.

Through trial and error, this research determined that a bin count of 5 provided a better rate of classification than bin counts greater or less than 5, such as 3 or 7. Bin counts above 7 became impractical for storage reasons with the larger subject programs. A bin count of 3 was ineffective and a bin count of 7 generally showed a zero to twenty percent degradation in the rate of classification over a bin count of 5. Thus, for each modeled variable in a program, a transition array of size 5x5 was used. A future research goal is to examine bin counts that are dynamically estimated for each variable as a way to find the optimal number of bins.

For this experiment, the instrumentation of the subjects was done using the Daikon invariant detector, described in Chapter 6. Daikon's front end for C, Kvasir, instruments and executes C programs using the DWARF-2 debugging format.[23] Kvasir produces trace files of the values assigned to variables. The collected data was then post-processed for use in the Argo infrastructure. This post-processing reduced the number of variables as described in Section 5.2.2 and then calculated the state-transition profiles between pairs of the five databins for each variable. The study proceeded using the same protocols as described in Study 2, with ten training epochs each incrementing the training set for the classifier by 25 instances.

Figure 7.2 summarizes as graphs the results for all subjects and their databin-transition classifiers evaluated for both batch learning and active learning. In this figure, there is one graph for each of the six studied subjects. Each graph is structured similarly to those in Figure 7.1 for Study 2. Only *flex* fails to meet the minimum of a classification rate of 0.5 for either batch or active learning, although the classifier at epoch 1 does perform slightly better than 0.5.

The active-learning classifiers for *space, replace,* and *print_tokens* compare favorably with the active-learning classifiers for branch profiles in Study 2. While this study demonstrates that a value-flow feature such as databin transitions can be used to build effective classifiers, another study is required to compare these results with those using control-flow features. Databins as implemented here are more expensive to collect because of the overhead of Daikon and the necessary post-processing. The next study compares the effectiveness of databin transitions with branch profiles and seeks to learn if they can be combined to advantage.

[23] http://pag.csail.mit.edu/daikon

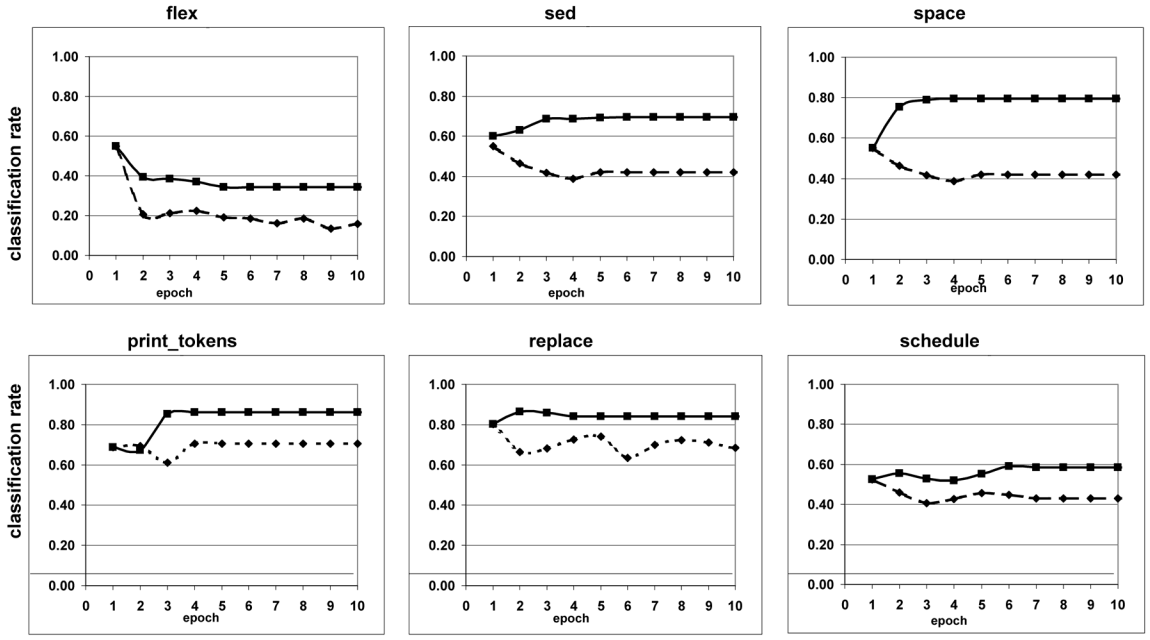


Figure 7.2. Comparison of batch and active learning for databin classifiers.

7.3.6 Study 4: Evaluating ensemble classifiers

The goal of this study was to evaluate whether the classification rates of ensemble classifiers composed of a control-flow classifier (branch profiles) and a databin-transition classifier were significantly different from the classification rates of the two component classifiers.

For this experiment, both the branch-profile and databin transition profile classifiers were trained again using active learning. The protocol, however, remained the same as in Studies 2 and 3, with 10 training epochs and increments of 25 training instances per epoch. The ensemble classifier, as described in Section 5.3, classifies using a simple weighted voting scheme [65]. The two component classifiers report their respective classification labels, including "unknown," and their confidence in these

labels. The voting scheme consists of comparing the labels reported by the two classifiers. If the two classifiers agree, the common label is used to label the execution. If they disagree, the classifier with the higher confidence wins the vote and its label is used to label the execution.

The classification results are shown in Figure 7.3 as six graphs each corresponding to one of the six subject programs. The dotted lines in each graph plot the databin-transition classifier while the dashed lines represent the branch profile classifiers. The ensemble classifier is plotted as a solid line. These graphs show that in all cases, except for subject *flex*, both component classifiers begin at a classification rate above 0.5. This corresponds to an error rate below 0.5, which is one of the necessary and sufficient properties for successful ensembles (see Section 5.3.) Note that the graphs show that in

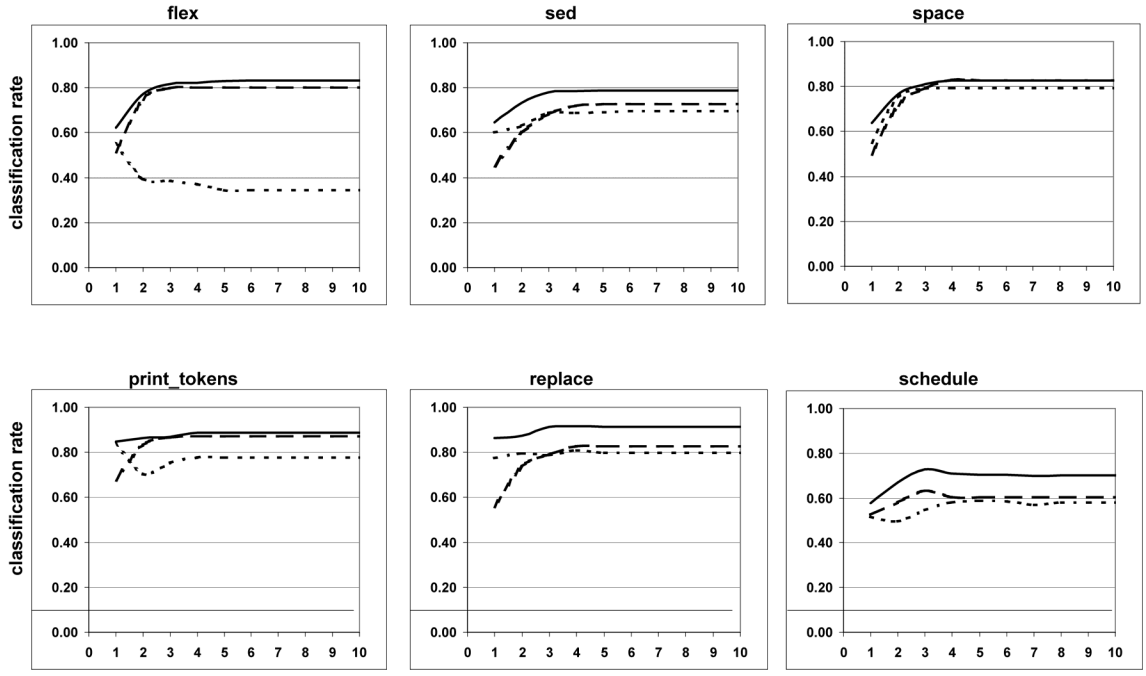


Figure 7.3. Ensembles (solid) of branch-profile (dashed) and databin-transition-profile (dotted) classifiers.

all cases, branch profiles do better than databin profiles. The graphs also show that the ensemble classifier is as good as or better than the branch profile classifier. To determine the statistical significance of these results, a two-sample t-procedure was performed on the means of the epoch 10 classifiers [29]. The branch-profile classifiers (BPC) were compared to the ensemble classifiers (EC) and the databin profile classifiers (DtPC) were compared with the ECs at epoch 10. The comparisons were done by testing hypotheses on the means of the distributions, using a two-sample t-procedure with $\alpha = 0.05$.[24] In both cases, the null hypothesis was that the means are equal:

$$H_0 : \mu_{EC} = \mu_{BPC} \text{ and } H_0 : \mu_{EC} = \mu_{DtPC};$$

the one-sided alternative hypotheses were:

$$H_a : \mu_{EC} > \mu_{BPC} \text{ and } H_a : \mu_{EC} > \mu_{DtPC}.$$

Table 7.3 presents the results. For each subject listed in the left-hand column, the columns show the sample size of epoch 10 classifiers (*n*); the sample mean and variance ($\bar{x}$ and $\sigma_{\bar{x}}$) for EC, BPC, and DtPC; the degrees of freedom (*df*) and corresponding critical t-value (*t-crit*); and the test statistic (*t-stat*), P-value, and hypothesis truth for each of the two null hypotheses. The last column reports whether EC's classification rate was a statistically significant improvement over the rates of both BPC and DtPC. For each subject except *space* and *print_tokens*, there is a strong rejection of both null hypotheses since the P-values are less than α. Therefore, the alternative hypotheses that the mean of the EC distribution is greater than the mean of either of the component (BPC or DtPC) distributions is supported. For these subjects, then, there is a statistically significant advantage to using ensemble classifiers to improve the rate of classification. For *space*,

[24] A two-sample t-procedure compares the means of two normal distributions when the variations are unknown. α is the level of significance.

Table 7.3. Epoch 10 two-sample t-procedure hypothesis tests, alpha = 0.05.

		EC		$ComponentClassifier$			$H_0 : \mu_{EC} = \mu_{Component}$					EC
Subject	n	$\bar{x}$	$\sigma^2_{\bar{x}}$	Name	$\bar{x}$	$\sigma^2_{\bar{x}}$	df	t-crit	t-stat	P-val	H_0	*Exceeds Both*
flex	30	0.833	0.003	*BPC*	0.800	0.003	58	1.672	2.445	0.009	Reject	Yes
				DtPC	0.330	0.021	36	1.688	17.767	0.000	Reject	
sed	22	0.787	0.020	*BPC*	0.728	0.004	29	1.699	1.792	0.042	Reject	Yes
				DtPC	0.696	0.034	39	1.685	1.835	0.037	Reject	
space	30	0.827	0.018	*BPC*	0.839	0.001	33	1.692	-1.633	0.056	Accept	No
				DtPC	0.794	0.015	58	1.672	1.004	0.160	Accept	
print_tokens	26	0.888	0.004	*BPC*	0.871	0.005	49	1.677	0.917	0.182	Accept	No
				DtPC	0.778	0.020	34	1.691	3.639	0.001	Reject	
replace	37	0.914	0.011	*BPC*	0.820	0.005	64	1.669	4.444	0.000	Reject	Yes
				DtPC	0.827	0.030	59	1.674	2.601	0.006	Reject	
schedule	37	0.702	0.036	*BPC*	0.604	0.037	72	1.666	2.190	0.016	Reject	Yes
				DtPC	0.580	0.068	66	1.668	2.290	0.013	Reject	

there is no statistical significance to using EC instead of BPC or DtPC. For *print_tokens*, there is no statistical significance to using EC instead of BPC, but there is a statistical significance to using EC instead of DtPC. It follows for the other subjects that the improvement in the EC over the two component classifiers represents the difference of their respective classifications.

7.4 Studies supporting example applications

The goal of this second set of empirical studies is to support the example applications presented in Chapter 6. In Section 7.4.1 there is an explanation of how the empirical studies in Section 7.3 support the two applications of automated behavior detection. Section 7.4.2 presents a case study to support the potential of parallelized debugging. Finally, Section 7.4.3 presents a case study exploring the use of motifs as classifiers.

7.4.1 Automation of behavior detection

The example applications of the two categories of use presented in Sections 6.1, "Software Testing," and Section 6.2, "Failure Detection in Deployed Software," are both supported by the four empirical studies presented above in Section 7.2

The core process in the application for automating test-suite augmentation relies on the effectiveness of active learning to automatically filter new executions of a program on the basis of behavior recognition. The studies in this chapter of the three control-flow classifiers, one value-flow classifier, and one ensemble classifier support the argument that active learning can refine a classifier so that its rate of classification is better than that for a batch learner trained from the same data. Thus, the conclusion is that for these subjects, this application is a good example of the potential usefulness of the techniques developed in this research.

The ability of the classifiers in the first set of studies to discern correctly "passing" from "failing" behaviors supports the application of these techniques to automatically detect new behaviors as well as "failing" behaviors from data collected in the field without knowledge of the inputs or outputs of the program being monitored. The machinery for active learning provides the detection capability for detecting "unknown" behaviors. The threshold for this detection, described in Section 3.3.3, is a tunable heuristic, so in this application, the developer could adjust the threshold, if needed, to fit a particular deployment paradigm.

An additional case study was performed to examine the classification process from a different perspective. In this study, six versions of *space* have known test suites in the ARG repository with an unusual property: the test suites are just large enough to

provide coverage of every executable branch yet they do not induce any failures. The goal of the case study was to determine whether a classifier trained on one of these branch-covering test suites would recognize or not recognize the known failing test cases.

The results are presented in Table 7.4. The three columns in Table 7.4 show the version of *space,* the number of failing test cases out of 13,585 test cases for each version, and finally the percent of these failing test cases that were deemed "unknown." As an example, version "v12" has 18 failing test cases available. When a classifier for this version was built from branch profiles using the branch coverage test suites, the classifier rated 52% of the eighteen failing executions as "unknown." The classifier was able to detect new behaviors in the set of failing tests. The failing test cases not labeled as "unknown" were therefore classified as "passing." Three of the six versions in Table 7.4 have recognition rates at or above 60%, including "v28" with 6704 failures. Version "v12" shows a rate above 50%. These classifiers were built using batch learning and in general, *space* showed a classification rate of 60% for batch learning regardless of training set size as shown in Table 7.2. The results of this study are consistent with the studies of *space* presented in Section 7.3. These results also suggest that the DTMCs built from branch profiles for these versions of *space* include information beyond simple branch coverage. The implication of this result for testing is that a branch-coverage

Table 7.4. Recognition rates of failing test cases.

version	failures	% unknown
v12	18	52
v18	18	33
v22	39	62
v27	19	0
v28	6704	60
v33	7	67

adequate test suite, such as those used in this study, may not detect all failing executions of a program that are related to the branching behavior of a program.

7.4.2 Parallelized fault localization

The goal of this empirical study was to explore the application of the techniques developed in this research to parallelized fault detection, as described in Section 6.3.

The subject for this study is *space* and, for this study, 71 new 8-fault versions of *space* were created as examples of a program containing multiple faults. Each of the eight faults in each version was selected from the ARG repository of single faults at random. Then, each version was instrumented first for statement coverage, which is the input data for *Tarantula*, and secondly for branch profiles, which is the input data for *Argo*. As described in Section 6.3, *Tarantula* is one of several systems that provide assistance in fault-localization by providing a ranked list of program statements. These lists rank statements by how likely they are to be associated with a fault.

The ARG repository contains 1,000 test suites for *space* that have been constructed to provide branch coverage. That means that every executable branch is executed at least once in every test suite. For the purposes of this study, each of these test suites was considered to be a reasonable surrogate for a real-world development effort that produced test suites that required branch coverage.

The basic empirical approach for each 8-fault version was to simulate the debugging environment for a developer by locating as many of the faults as possible using each of the two protocols under study: sequential and parallel fault-localization.

The measures chosen were cost, iterations, and number of faults. These measures are defined below.

7.4.2.1 Cost

The cost for locating a fault is expressed as the relative depth in the list of ranked statements that *Tarantula* produces. If no tests fail, *Tarantula* cannot produce suspicious statements. The full number of ranked statements is by definition the number of executable statements in the program. For *space*, this is approximately 3,660. For example, if the first fault is found within the top 100 suspicious statements, then the *cost* is this reported percentage, which is (100 / 3660) * 100 = 2.73%. This percentage of source code that is examined is a simple proxy for the cost of the engineering effort expended in locating the fault.

7.4.2.2 Iterations and number of faults

The *iterations* measured are the number of times that the program is compiled and tested to locate faults. For the sequential protocol, if all eight faults are found, then there will be eight iterations. It is possible that one or two faults will not be exposed because the test suite does not contain any inputs that exercise the fault. In this case, for the sequential protocol, the number of faults that are exercised will be equal to the number of iterations to find them all.

Each 8-fault version was studied independently of the others. For each 8-fault version *8-F*, one test suite was chosen at random and was kept throughout the fault-localization process for that 8-fault version until every fault that was exercised by the test

suite was located.[25] The entire fault-localization process was conducted for *8-F* once for the sequential protocol and once for the parallel protocol.

In the sequential protocol, *8-F* was run against the selected test suite *T*. *Tarantula* produced a ranked list of statements and from it, *Tarantula* calculated a cost to find the first fault and also identified this fault. Then the fault was removed from the code and *space* was recompiled and re-tested against *T*. This process repeated sequentially until the test suite no longer induced any failures. The total cost for this sequential process to locate all the locatable faults is the sum of the costs from each iteration.

The results from the sequential study are shown in Table 7.5. The left-most column lists the 8-fault versions. For historical reasons, the individual faults are known as "v"*n*, where *n* is a fault number and thus, fault number 10 is known as *v10*. The subject names are composed of the names of the 8 faults in numerical order separated by an underscore. The center group of four columns is labeled "sequential." Within this group are the columns for cost, iterations, number of faults, and cost per fault. For example, the first subject is v10_v12_v17_v18_v20_v21_v27_v30 and it had a total cost of 30.35 in 8 iterations to find 8 faults. The cost per fault is then 30.35 / 8 = 3.79. The cost per fault translated to lines of code is (3.79 / 100) * 3200 = 121 lines of code examined per fault located. The averages for all measures are shown in the last row.

In the parallel protocol, *8-F* was run against the selected test suite *T*. This time, data was collected for *Argo* from branch profiles as well as for *Tarantula* from statement execution. *Argo* then modeled all of the failing executions as DTMCs based on branch profiles. The goal was that automated clustering would yield clusters that focused the faults so that the failing executions within each cluster would be primarily associated

[25] Future research with these subjects will include a larger sample of the test suites.

Table 7.5. Sequential vs. parallel costs for locating faults in 8-fault versions of *space*.

	SEQUENTIAL				PARALLEL			
8-fault version of space	cost	iterations	num faults	cost per fault	cost	iterations	num faults	cost per fault
v10_v12_v17_v18_v20_v21_v27_v30	30.35	8	8	3.79	23.33	4	8	2.92
v10_v13_v15_v19_v24_v27_v33_v38	16.90	8	8	2.11	13.78	3	8	1.72
v10_v14_v16_v17_v28_v29_v34_v36	107.38	8	8	13.42	153.90	4	8	19.24
v10_v14_v24_v25_v26_v31_v35_v36	30.36	8	8	3.80	20.67	3	8	2.58
v10_v16_v20_v25_v27_v28_v36_v38	37.64	6	6	6.27	39.09	3	6	6.52
v11_v12_v13_v17_v19_v21_v33_v36	22.01	6	6	3.67	15.88	2	7	2.27
v11_v13_v16_v19_v20_v23_v25_v29	42.92	8	8	5.36	45.57	3	8	5.70
v11_v16_v17_v18_v20_v21_v34_v35	13.49	6	6	2.25	7.42	3	6	1.24
v11_v16_v20_v24_v31_v33_v34_v38	18.31	7	7	2.62	17.44	2	7	2.49
v12_v14_v15_v16_v20_v30_v33_v34	10.25	7	7	1.46	18.70	5	7	2.67
v12_v20_v23_v28_v29_v33_v36_v38	68.94	8	8	8.62	121.76	3	8	15.22
v13_v20_v23_v28_v31_v33_v34_v35	49.56	6	6	8.26	24.61	3	7	3.52
v14_v15_v18_v20_v21_v27_v34_v37	5.08	5	5	1.02	23.65	4	5	4.73
v16_v17_v20_v21_v22_v23_v27_v37	26.47	8	8	3.31	18.55	4	8	2.32
v3_v10_v12_v13_v24_v25_v34_v36	27.47	6	6	4.58	142.52	2	6	23.75
v3_v10_v21_v24_v25_v30_v34_v38	38.08	7	7	5.44	17.58	4	7	2.51
v3_v11_v13_v17_v28_v30_v31_v33	41.62	8	8	5.20	28.22	4	8	3.53
v3_v12_v13_v15_v17_v19_v22_v30	42.34	8	8	5.29	22.83	3	8	2.85
v3_v12_v13_v21_v22_v23_v27_v34	24.52	5	5	4.90	61.88	2	5	12.38
v3_v12_v13_v22_v24_v25_v30_v31	30.42	7	7	4.35	93.06	3	7	13.29
v3_v14_v21_v23_v24_v31_v36_v37	51.36	8	8	6.42	39.96	3	8	5.00
v3_v4_v13_v16_v18_v27_v28_v34	63.45	6	6	10.57	29.31	4	6	4.88
v3_v4_v14_v20_v22_v25_v26_v28	78.85	8	8	9.86	66.12	5	8	8.26
v3_v4_v19_v21_v23_v24_v26_v27	44.79	8	8	5.60	15.23	3	7	2.18
v3_v4_v8_v21_v22_v24_v31_v33	40.92	8	8	5.11	57.01	2	8	7.13
v3_v5_v6_v12_v22_v26_v35_v36	31.80	7	7	4.54	98.58	3	7	14.08
v3_v5_v7_v12_v13_v19_v29_v36	67.38	8	8	8.42	113.49	3	8	14.19
v3_v5_v9_v19_v24_v29_v30_v34	98.00	8	8	12.25	145.86	4	8	18.23
v3_v6_v10_v23_v24_v27_v33_v35	47.36	8	8	5.92	49.53	1	6	8.26
v3_v6_v7_v11_v14_v15_v18_v27	33.59	6	6	5.60	39.74	3	6	6.62
v3_v6_v7_v12_v18_v20_v28_v29	85.21	8	8	10.65	114.91	3	8	14.36
v3_v6_v7_v13_v16_v18_v29_v36	70.13	7	7	10.02	112.37	3	7	16.05
v3_v6_v7_v15_v22_v24_v29_v37	79.22	8	8	9.90	50.26	4	8	6.28
v3_v6_v8_v10_v11_v12_v19_v21	41.20	7	7	5.89	74.19	2	7	10.60
v3_v7_v12_v17_v20_v27_v36_v38	48.60	7	7	6.94	92.11	3	7	13.16
v3_v7_v13_v23_v25_v31_v33_v37	34.36	7	7	4.91	31.44	2	7	4.49

Table 7.5. (Continued).

8-fault version of space	SEQUENTIAL cost	iterations	num faults	cost per fault	PARALLEL cost	iterations	num faults	cost per fault
v3_v7_v15_v22_v24_v26_v28_v38	66.73	8	8	8.34	66.09	3	8	8.26
v3_v7_v16_v18_v19_v22_v34_v38	27.96	6	6	4.66	42.80	3	6	7.13
v3_v8_v10_v18_v30_v31_v33_v37	22.45	6	6	3.74	84.59	3	6	14.10
v3_v8_v15_v18_v20_v21_v23_v24	43.74	7	7	6.25	33.08	4	7	4.73
v3_v8_v9_v17_v20_v22_v25_v26	83.79	8	8	10.47	70.47	3	8	8.81
v3_v9_v11_v16_v17_v20_v24_v27	60.20	8	8	7.52	114.04	4	6	19.01
v3_v9_v12_v24_v25_v31_v36_v37	53.39	8	8	6.67	91.94	3	8	11.49
v4_v10_v11_v19_v27_v31_v36_v37	15.34	7	7	2.19	13.66	3	7	1.95
v4_v12_v14_v18_v21_v34_v36_v38	12.25	5	5	2.45	13.50	2	5	2.70
v4_v12_v18_v19_v26_v27_v30_v37	8.28	7	7	1.18	4.54	5	7	0.65
v4_v5_v14_v15_v17_v22_v25_v27	22.44	7	7	3.21	28.69	3	7	4.10
v4_v5_v7_v17_v18_v25_v27_v29	76.98	8	8	9.62	84.07	4	8	10.51
v4_v5_v7_v9_v17_v25_v26_v35	50.22	8	8	6.28	49.45	3	8	6.18
v4_v5_v9_v10_v16_v30_v31_v38	52.21	8	8	6.53	27.77	4	8	3.47
v4_v5_v9_v16_v27_v30_v31_v35	22.89	7	7	3.27	23.08	3	7	3.30
v4_v6_v12_v18_v19_v27_v29_v35	41.76	7	7	5.97	35.74	4	7	5.11
v4_v6_v12_v24_v25_v26_v30_v36	21.64	7	7	3.09	18.68	6	7	2.67
v4_v6_v16_v20_v25_v27_v33_v38	14.95	7	7	2.14	9.29	4	7	1.33
v5_v10_v14_v15_v17_v26_v34_v35	26.00	7	7	3.71	25.50	3	7	3.64
v5_v10_v18_v29_v30_v33_v34_v36	70.48	6	6	11.75	131.85	4	6	21.97
v5_v12_v13_v14_v21_v23_v31_v35	11.57	7	7	1.65	10.62	2	7	1.52
v5_v12_v13_v16_v19_v22_v23_v35	7.12	7	7	1.02	8.37	3	7	1.20
v5_v14_v16_v19_v22_v30_v33_v36	8.72	7	7	1.25	18.71	2	7	2.67
v5_v16_v19_v27_v31_v34_v36_v38	7.65	5	5	1.53	2.90	3	5	0.58
v5_v6_v7_v12_v16_v18_v24_v30	21.11	6	6	3.52	11.70	3	6	1.95
v5_v6_v7_v13_v18_v21_v28_v33	38.15	7	7	5.45	24.30	5	7	3.47
v5_v6_v9_v11_v12_v14_v25_v37	39.23	7	7	5.60	18.97	4	7	2.71
v5_v6_v9_v12_v16_v21_v28_v31	52.68	7	7	7.53	45.79	4	7	6.54
v5_v7_v14_v15_v20_v21_v26_v31	13.54	8	8	1.69	19.75	5	8	2.47
v5_v7_v14_v15_v22_v28_v30_v36	9.72	8	8	1.22	34.77	6	8	4.35
v5_v7_v8_v15_v20_v23_v31_v37	15.14	8	8	1.89	13.09	3	8	1.64
v5_v8_v10_v14_v22_v25_v28_v36	56.85	8	8	7.11	37.42	5	8	4.68
v5_v8_v10_v15_v22_v24_v25_v38	19.73	8	8	2.47	16.15	4	8	2.02
v5_v9_v12_v18_v20_v26_v31_v37	34.01	8	8	4.25	19.25	4	8	2.41
v5_v9_v15_v21_v22_v25_v26_v27	32.69	8	8	4.09	6.21	3	7	0.89
v6_v11_v14_v23_v29_v31_v36_v37	60.83	8	8	7.60	48.00	4	8	6.00

Table 7.5. (Continued).

8-fault version of space	SEQUENTIAL cost	iterations	num faults	cost per fault	PARALLEL cost	iterations	num faults	cost per fault
v6_v11_v17_v21_v22_v27_v33_v38	16.14	7	7	2.31	11.17	3	7	1.60
v6_v11_v17_v24_v27_v29_v34_v38	94.21	8	8	11.78	132.48	3	8	16.56
v6_v12_v17_v18_v30_v31_v34_v35	17.82	5	5	3.56	16.30	3	5	3.26
v6_v21_v22_v23_v33_v35_v36_v37	13.14	7	7	1.88	8.83	5	7	1.26
v6_v7_v15_v22_v24_v31_v33_v36	23.29	8	8	2.91	25.23	6	8	3.15
v6_v7_v8_v10_v11_v28_v34_v36	42.89	7	7	6.13	33.88	5	7	4.84
v6_v7_v8_v19_v21_v24_v25_v38	15.65	8	8	1.96	9.96	5	8	1.25
v6_v7_v9_v20_v25_v33_v34_v35	34.72	6	6	5.79	16.31	4	6	2.72
v6_v8_v10_v11_v16_v18_v22_v37	18.03	7	7	2.58	18.31	3	7	2.62
v6_v8_v13_v20_v21_v27_v36_v38	7.17	5	5	1.43	3.80	2	5	0.76
v6_v9_v16_v24_v26_v35_v36_v38	35.57	8	8	4.45	24.20	4	8	3.02
v7_v10_v12_v17_v22_v26_v27_v37	35.97	8	8	4.50	17.91	5	8	2.24
v7_v10_v16_v17_v18_v31_v33_v37	24.92	6	6	4.15	20.06	2	6	3.34
v7_v10_v21_v24_v26_v27_v31_v37	19.99	8	8	2.50	9.02	3	8	1.13
v7_v13_v15_v16_v20_v25_v34_v38	12.78	7	7	1.83	16.58	3	7	2.37
v7_v15_v20_v23_v24_v26_v27_v31	18.35	8	8	2.29	18.72	3	8	2.34
v7_v8_v10_v18_v20_v21_v23_v26	17.19	8	8	2.15	13.36	4	8	1.67
v7_v9_v18_v19_v28_v31_v34_v38	55.75	6	6	9.29	26.41	4	6	4.40
v7_v9_v21_v23_v24_v25_v30_v34	18.50	7	7	2.64	13.98	5	7	2.00
v8_v10_v12_v14_v18_v24_v25_v33	24.86	6	6	4.14	15.95	2	6	2.66
v8_v10_v24_v25_v27_v31_v34_v38	16.33	7	7	2.33	12.27	3	7	1.75
v8_v12_v14_v18_v20_v22_v26_v36	13.38	8	8	1.67	15.69	3	8	1.96
v8_v13_v17_v19_v25_v26_v30_v37	22.24	8	8	2.78	26.68	4	8	3.33
v8_v15_v22_v23_v29_v30_v34_v35	64.13	8	8	8.02	67.04	4	8	8.38
v9_v10_v14_v21_v22_v24_v33_v34	24.63	7	7	3.52	27.93	3	6	4.66
v9_v10_v19_v22_v23_v33_v35_v37	37.21	8	8	4.65	26.50	2	8	3.31
v9_v12_v13_v14_v16_v17_v36_v37	52.86	8	8	6.61	32.98	4	8	4.12
v9_v12_v14_v18_v20_v31_v35_v37	25.57	6	6	4.26	12.56	3	6	2.09
Averages	**36.26**	**7.18**	**7.18**	**4.97**	**40.19**	**3.43**	**7.13**	**5.64**

*Relative time to find all faults = iterations * cost per fault*	*7.18 * 4.97 = 35.68*	*3.43 * 5.64 = 19.35*

with a specific fault. *Argo* was used to cluster the faults using the TRAINCLASSIFIER ALGORITHM and the *Knee* function to detect the stopping point. Careful examination of the clustering results showed that there was a clustering level that did focus the faults but that the stopping function failed to stop the clustering at this point. There were two basic cases of this mismatch. In the first case, the clustering stopped with too many clusters and it possible that the clustering algorithm detected sub-behaviors within one or more faults. In the second case, the clustering stopped with too few clusters and it is likely that the clustering algorithm saw the data from two or more faults as very similar.

To find an appropriate stopping criterion for the clustering process, a heuristic was developed. The first step in this heuristic is to fix the stopping criterion at one so that the clustering runs until there is only one cluster. At each clustering iteration, the membership of the clusters is recorded, so that the clustering history can be revisited. Thus, after the clustering completed, level 1 contains one cluster, and level 2 contains two clusters, etc. Each higher clustering level represents an earlier stage in the clustering history and therefore contains one cluster more than the previous level. To detect the appropriate stopping point this heuristic approach starts with level 2, and *Tarantula* produces a ranked list of statements for both clusters at level 2. Each of the two ranked lists is compared with the ranked list of the cluster at level 1, known as the *parent* cluster. The method of comparison is the Jaccard measure.[26] If at least one of the two clusters differs by a threshold from the parent cluster, then the process repeats by moving to

[26] The Jaccard similarity coefficient is a common statistic used to compare the similarity and diversity of data sets. It is defined on two sets S1 and S2 as the size of the set intersection of S1 and S2 divided by the size of their set union. For this study, the sets used were the top 20% of the statements in the ranked statement list produced by *Tarantula*. This proportion was determined empirically as a viable heuristic. Furthermore, the threshold used was also heuristically determined as a Jaccard similarity coefficient of 0.68.

cluster level 3. At level 3, there are three clusters, but only two of them were merged during the clustering process. The ranked statements of these two clusters at level 3 are each compared, using the Jaccard method, with the ranked statements of the parent cluster at level 2 that was formed by merging them. This comparison detects whether the merging of clusters mixed important information about different faults as represented by the sets of ranked statements. This process of evaluating ever-higher clustering levels continues until the Jaccard differences produced are less than the threshold. When they reach this point, the clustering stops. At this clustering level, each of the clusters of failing executions is assumed to focus one fault.

This resultant set of clusters is at the heart of the parallelized protocol. Each cluster was assumed to reveal one fault. At the first iteration, any number of faults might be found from one fault to eight faults (for these 8-fault versions.) The primary reason for this is that certain faults seem to dominate other faults by causing a failure before the dominated faults can even be exercised. For this study, *Tarantula* calculated the cost and the fault found for each cluster. The cost for the iteration is the sum of the costs for each fault. For example, if two clusters occurred at the first iteration and exposed two faults with costs *c1* and *c2*, the cost for the iteration is *c1* + *c2*. If both clusters exposed the same fault, the cost is still this sum, because this sum represents the effort expended to locate the faults. Then, all of the located faults are found and a new version of the subject, in this case a 6-fault version (formed by removing two faults from an 8-fault version), is created, instrumented, compiled, and run against *T*. This process repeats until no there are no failing test cases.

The results of the parallelized protocol are shown in Table 7.5 as the right-hand grouping of columns with the same column labels as those for the sequential results. The average cost per fault for the sequential protocol is 4.97 and for the parallelized protocol, it is 5.64. However, the average number of iterations for the sequential protocol is 7.18 and for the parallelized protocol, it is 3.43. Note that both protocols find an average of just over seven faults. The sequential protocol takes approximately twice as many iterations to complete the fault-finding task as does the parallelized protocol. This reduced number of iterations to find all the faults is the main advantage of the parallelized protocol over the sequential protocol.

To compare these two protocols, this research defines a simple cost model that says that one measure t of the time to find all the faults in a program is the number of iterations times the average cost per fault. This arises from the assumption that there are enough engineers available at each iteration to locate the faults. This model ignores all other costs, for example, the costs of running the test suite at each iteration. The calculations for t are shown at the bottom of the second page of Table 7.5. For the sequential protocol, this unit-less measure of time $t = 7.18 * 4.97 = 35.68$, and for the parallelized protocol, $t = 3.43 * 5.64 = 19.35$. In terms of this cost model, the parallelized protocol completes the same job in approximately half the time, but with more total effort because the average cost per fault is higher than for sequential. This cost model may be appropriate for those developers who are constrained to produce fault-free software in the shortest possible time.

7.4.3 Motifs

The goal of this case study was to determine whether motifs in traces of method calls recorded during program execution could be used to detect "passing" and "failing" behaviors. There exist several motif-finding software packages in the public domain and this research chose to use the MEME/MAST System.[27]

The MEME portion of this software package is designed to detect motifs in a set of protein sequences. To use this software required a data transformation. The protein alphabet is 20 characters long, and each of the Siemens programs has 20 or less methods, so the first transformation was to create a mapping from method name to alphabet letter. Each of the Siemens subjects was instrumented to produce a trace of method calls in the form of method names. These traces were then processed to remove all repeating method names and all repeating pairs of method names. This pruning of the traces eliminated many very simple motifs that were found to hide more meaningful motifs in initial explorations. Then each trace was transformed into a string of single characters using the mapping from method names to protein letters. Further minor textual manipulations were required to arrive at a format that the MEME program would accept as a set of protein strings. The other required parameters were the number of motifs to find and a limit on their size in terms of letters from the protein alphabet.

In this study, each version of the subjects was processed five times. Each time, 25 failing test cases were chosen at random from the failing test cases for that program version and became the training set. The traces from these 25 failing exectutions were transformed into the protein sequence alphabet and input to the MEME software. The maximum number of motifs was specified as 3 and the width as between 5 and 15 letters.

[27] http://meme.sdsc.edu/meme/intro.html

The MEME software returned an html file with extensive analysis of the motifs. This research proceeded by extracting automatically the three motifs most commonly found across all 25 submitted strings.

These three motifs were the components of a behavior classifier that could distinguish between "passing" and "failing" behaviors. By observation, it was determined that the most effective classifier architecture for these subjects was the disjunction of all three motifs in a trace. For example, if the motifs were *M1, M2,* and *M3*, then the presence of any of them in a trace signaled a "failing" execution and the absence of all of them in a trace signaled a "passing" execution.

The results of this case study are shown in Table 7.6. In the table, the subjects are listed in the left-hand column. The classification rate for "passing" executions is shown in the center column and the classification rate for "failing" executions is shown in the right-hand column.

Every classification rate shown in the table exceeds 50% and some are as high as 100%. Thus, it is clear that motifs of method calls have potential as the basis for classifier architecture for these subjects. This research also explored the location in the trace at which the classifier made its decision. For example, if any of the three motifs were found before the end of a trace, then this would be evidence that a "failing" execution was underway. Exploratory investigations show that location of the signal motif is rarely at the end of the trace, but often near the end of the trace. This result hints at the possibility of live detection of behavior. When the same experiment was performed by training on "passing" executions, the classification rate for detecting

Table 7.6. Case study of motifs-based classifier.

subject	classification rate for "passing"	classification rate for "failing"
print_tokens	79	75
replace	63	58
schedule	53	70
tcas	75	100
tot_info	83	70

"failing" executions were usually "0." The reason is that the discovered motifs are common to both "passing" and "failing" executions. Further research into building classifiers in this way is warranted by these results.

7.5 Threats to Validity

Although these empirical studies provide evidence of the potential usefulness of the techniques developed in this research, there are several threats to the validity of the results that should be considered in their interpretation.

Threats to the external validity of an experiment limit generalizing from the results. The primary threats to external validity for these studies arise because only a small set of C programs has been considered. Thus, this research cannot claim that these results generalize to other programs. In particular, no generalization can be made as to the effectiveness of either control-flow or value-flow features as predictors of behavior. However, there is a variety of subjects used in this research and therefore, these subjects are useful for exploring the techniques developed and evaluated.

Threats to the internal validity of an experiment occur when there are unknown causal relationships between the independent and dependent variables. In these studies, one threat to internal validity is the limited number of test cases for each subject. The collective properties of the test suites may affect the classification rates. However, these test suites have been developed to meet basic testing criteria such as statement and branch coverage and as such can be considered to represent a diversity of possible inputs for each program. A second threat is that only one clustering algorithm was used and that the researcher's subjective judgment determined the similarity function chosen. However, the clustering algorithm was shown to be very effective for these subjects. A third threat arises from the specification of databins, where all variables were constrained to five databins. However, by sampling results, it was found that five databins had significantly better predictive power than either three or seven bins. A final threat also involves databins and arises from the use of a heuristic to reduce the number of variables considered. However, the reduction makes the analysis tractable and the resulting classifiers were effective for the studied subjects.

Threats to construct validity relate to how the dependent variable, the classification rate, was produced. The implementation of *Argo* is complex and may contain errors, for example. However, Argo was carefully tested and the results spot-checked routinely, so there is reason to depend on the results of Argo. Another threat to construct validity is the size and variety of the test suites used for each subject. Some of the test suites were randomly generated, but others were created to meet certain testing criteria. The classification rate produced by classifying these test suites may be affected by the distribution of behaviors in the test suites. However, the reported efforts of the

researchers creating these test suites was to create a diversity of behaviors and this is demonstrated by the construction of effective classifiers from these test suites.

8 CONCLUSIONS AND FUTURE WORK

This dissertation presents an approach and a set of techniques that together comprise a first step towards laying the foundations of a rigorous solution to the problem of specifying, modeling, and implementing software self-awareness. Specifically, the approach and techniques enable the modeling and prediction of two external behaviors of a program ("passing" and "failing") using statistical summaries of data collected during the program's execution. The motivation for this research was to understand how a software gimbal might be constructed. The research conducted began to lay the foundational groundwork for the gimbal by exploring the properties of four individual stochastic features of program executions.

Four main contributions have been made by this work. First, this work demonstrates that features of program execution data can be modeled as stochastic processes that exhibit the Markov property and presents techniques for the automated construction of effective behavior classifiers based on these features for detecting "passing" and "failing" executions. Second, this work produced a prototype software gimbal that supports the modeling of execution data and the construction and evaluation of classifiers built from these models. Third, this work shows that the techniques developed are potentially useful in four categories of software engineering applications: software testing, failure detection, fault-localization, and software self-awareness. Finally, this work presents two sets of empirical studies that validate the thesis and support the use of the developed techniques in the example applications.

There are several areas of future research that this work suggests. One such area is the investigation of the potential mapping from the clusters of executions that are formed within each classifier to more specific behaviors than "passing" and "failing." Clearly there are sub-behaviors within each of these groups because software is generally developed from a set of behavioral specifications or requirements. One approach is to group executions by their known relation to some requirement or other and then to train the classifiers using the supervised learning stage to recognize each specified behavior. Will such training improve the quality of the classifications? A related research question is whether the sub-behaviors that cluster in the unsupervised learning stage map specifically to a requirement or set of requirements. In a preliminary investigation of the subject, *space*, twenty-seven distinct classes of behavior were identified within the "passing" behavior of a fault-free version based on the presence of keywords in the output files. When classifiers were trained on samples of each labeled behavior, the classifiers were able to correctly label new executions with a rate approaching 1.0. This study invites further investigation into sub-behaviors.

Another area of future research is to determine how to refine the studied features as well as other features so that their state spaces are as small and efficient as possible while still remaining effective. A related question is whether some features are better-suited than others to describing the execution of specific types of programs or even procedures within programs. If so, then there will be need for a calculus to describe how to choose and combine these features to the best effect.

There is potential research also involving the more global aspects of software self-awareness. One question, for example, is how should the software development process

and possibly software itself change to accommodate the implementation of effective software self-awareness. The abstractions offered by the Unified Modeling Language and by Model-Driven Architecture are potential starting points for approaching this problem. For example, event transitions occur in these models just as they do in executing programs and the techniques developed in this research may find applicability to these transitions as well.

Finally, there are likely other categories of use for these techniques. For example, how might these techniques help to provision software in a remote location with the capacity for self-assessment and possible recovery from difficult circumstances?

APPENDIX A

ARCHITECTURE OF ARGO

The first phase of the prototype software gimbal developed in this research is named Argo.[28] This appendix summarizes the architecture of Argo and then sketches the principal abstract classes that form the design and implementation of this architecture.

ARGO_BehaviorModel abstract class

The basic element is the *ARGO_BehaviorModel*, shown in the class diagram in Figure A.1. This class is presented in detail because its design illustrates the design of Argo. In general, this implementation uses the notation of underscore ("_") followed by the attribute name for a private field. A class *property* for a field accesses the field by subsuming a getter and a setter method. The property name is simply the attribute name without the leading underscore. Thus, for example, *UniqueID* serves, depending on the use context, as either a setter or a getter for *_UniqueID*.

In this class, *UniqueID* is a static field to track all instances of this class, while *ModelName* and *ModelID* are identifiers available to all members. The assumption here is that the models capture state transition profiles and these are stored in a jagged array[29]

[28] The name evolved from my story: I spent six months creating visualizations of program behaviors in 2003 and realized that I did not know enough about program behaviors to reify them graphically and convincingly. At the same time I read a detective story, "The Shape of Water [24]," that inspired me to think about the shape of running programs which, like water, are also considered amorphous. However, vessels and especially ships give shape to water and I turned to Western mythology for a name that captured the essence of this concept. In Greek mythology, the Argonauts sailed under the leadership of Jason to find the Golden Fleece in a ship they build and named "Argo." I borrowed this name for the framework that would help me to give form to the behaviors of software. In a fitting coincidence, my advisor had previously named her research group at Georgia Tech the Aristotle Research Group, the initials of which form the first three letters of Argo.

[29] A jagged array is a representation of a sparse matrix. Here [][] means that each row has its own size.

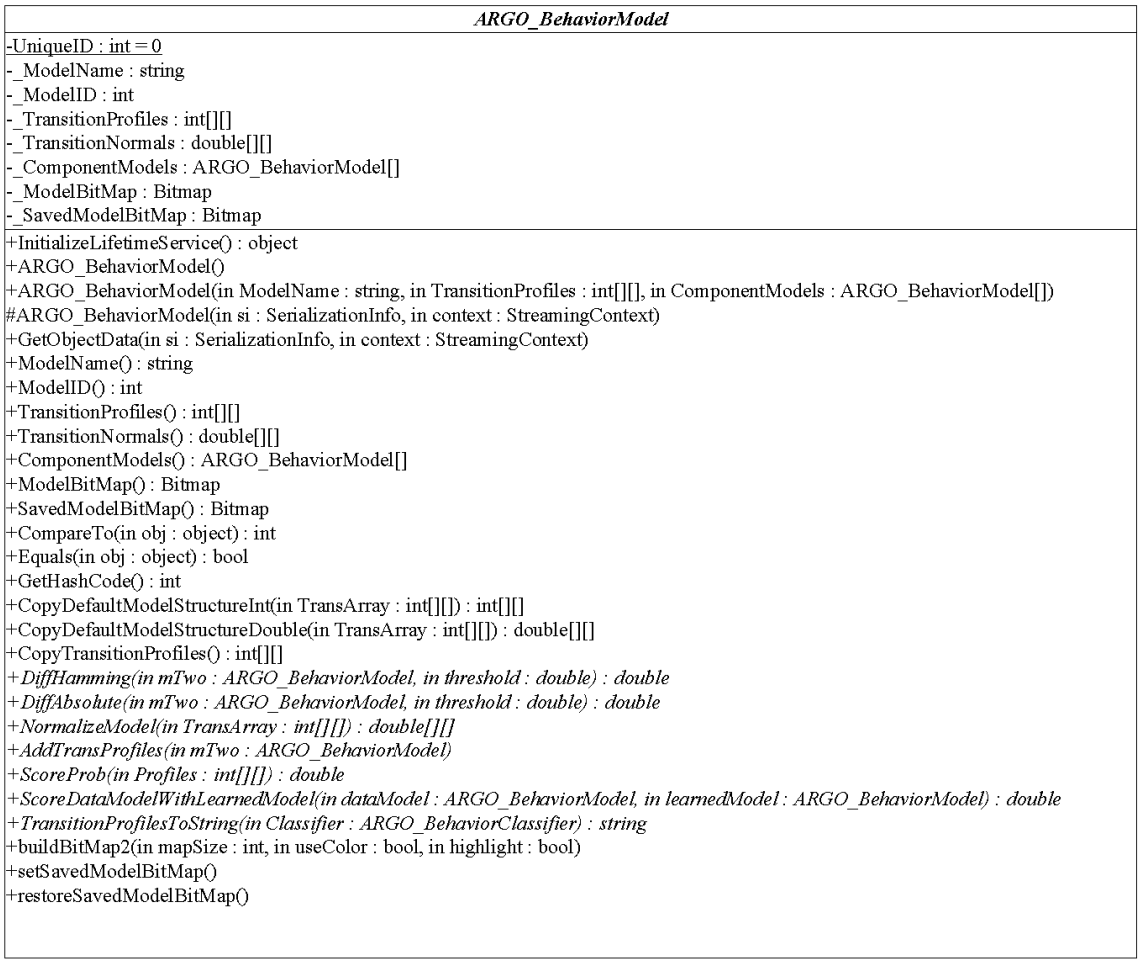


Figure A.1. Class diagram of ARGO_BehaviorModel.

TransitionProfiles. There is a second jagged array, *TransitionNormals,* that stores the row-normalized values of the transition profiles, representing a Markov chain. Because the attribute *ComponentModels* is an array of objects of type *ARGO_BehaviorModel,* any object of type *ARGO_BehaviorModel* can represent recursively a cluster of models. The two transition arrays then summarize this cluster, which itself contains other models each with its own summary. The attributes *ModelBitMap* and *SavedModelBitMap* store visual representations of the model.

This abstract class is powerful and adaptable to the research needs described herein. Note that while it does assume a finite set of transition profiles, it contains no direct reference to any actual states, as discussed below. This class implements three dotNet interfaces: *MarshalByRefObject, IComparable*, and *ISerializable*. These interfaces respectively provide for transport, comparison, and persistent storage.

The methods of this class comprise five main groups: constructors (including serialization and deserialization), properties (getters and setters), definitions for comparison (*CompareTo*, *Equals*, and *GetHashCode*), utility methods for copying arrays, and abstract specifications for required methods. This last group of abstract specifications is discussed next. *AddTransitionProfiles* (*ARGO_BehaviorModel mTwo*) : *void* specifies an implementation that adds the elements of the *TransitionProfiles* array of the invoking object and of a second object of this type. Thus, the implementation must check the shapes of the arrays and specify where the result is stored, for instance. *NormalizeModel (int [][] TransArray) : double [][]* specifies that the implementation of this class provide a way to normalize a jagged array. This research requires specific

techniques for comparing models as an aid to automated clustering. This abstract class specifies two techniques: the Hamming distance

(*DiffHamming (ARGO_BehaviorModel mTwo, double threshold) : double*),

and the absolute distance

(*DiffAbsolute (ARGO_BehaviorModel mTwo, double threshold) : double*),

both of which compare the *TransitionNormals* arrays of two models. While this abstract class does not provide the mechanics, it does specify a threshold value to use in the calculations. In general, the Hamming distance between two binary numbers is the count of bits in which they differ. The intuition here for a Hamming distance between two *ARGO_BehaviorModel*s is that since each cell of the *TransitionNormals* arrays contains by definition a value between 0 and 1, that each cell could be converted to a 0 or a 1 using a threshold value. The resulting arrays then map directly to two binary numbers whose digits are the contents of the array cells. The Hamming distance between these two binary numbers is easily calculated. Similarly, the absolute difference between two *TransitionNormals* arrays is the sum of absolute cell-by-cell differences of the two arrays. Again, the implementation of the abstract class must enforce the compatibility of the models.

Finally, the abstraction specifies two scoring methods:

ScoreProb (int[][] Profiles) : double and

ScoreDataModelWithLearnedModel (ARGO_BehaviorModel dataModel,

ARGO_BehaviorModel learnedModel) : double.

Scoring in this context is a specific mathematical calculation that produces the probability, or the equivalent negative logarithm of the probability, that a given Markov

model produced a specific sequence of states as summarized by a transition matrix [35]. *ScoreProb* scores directly its argument *int[][] Profiles* using the calling object's *TransitionNormals* array as the Markov model. This intentionally leaves the implementation to the concrete class. *ScoreDataModelWithLearnedModel* requires an implementation that is more flexible in that it can score any instance of *ARGO_BehaviorModel* with any other. This method is used in this research for scoring the models in a testing set with the models that comprise a classifier.

Argo class hierarchy

This section describes the abstract class hierarchy of the Argo framework with an emphasis on design and with less detail than in the previous section. The architecture describing program behaviors has three main levels, with *ARGO_BehaviorModel* at the bottom. This hierarchy of abstract classes is the heart of the Argo framework. Of course, there are many other classes for representing and manipulating subjects, but they are tangential to this core hierarchy. An implementation of this hierarchy can collect a set of transition profiles, model them individually, build a classifier from a training set chosen from these models, and use the classifier to classify the training set or a testing set of similar models.

The top level is the *ARGO_BehaviorClassifier*, which stores information about the subject program as well as stores a simple representation of the inter-procedural control-flow graph of the subject. The behavior classifier also stores a master map of the states and corresponding template for the jagged arrays that it will supply to any

individual behavior models that it creates. The behavior classifier maintains an array of *ARGO_Behavior-Specification*s.

The middle level is the *ARGO_BehaviorSpecification*. The behavior specification contains an array to store its component execution models (each a behavior model), an array of clusters (each also as a behavior model), a behavior label for this set of executions, and a reference to a training technique such as agglomerative hierarchical clustering. The operative mechanism for modeling program executions within this framework is that the subject program is instrumented to produce a set of state transition profiles for each execution. These profiles together with a map of the states, such as method names, and the behavior labels, if known, are the raw data for this modeling process. Each level of this hierarchy maintains some portion of this information. For example, the behavior models store only the specific profiles for individual executions as well as the cumulative profiles for a cluster of models. The behavior specification contains an array all the individual behavior models representing each execution that is part of the particular specification. The behavior specification also contains an array of behavior models, each of which is a cluster of behavior models that is the work product of the training method.

REFERENCES

[1] *Proceedings of the International Workshop on Self-Repairing and Self-Configurable Distributed Systems, (RCDS 2002)*: IEEE Computer Society Technical Committee on Distributed Processing, 2002.

[2] "Perpetual testing," http://www.ics.uci.edu/~djr/edcs/, date accessed Nov 2003.

[3] *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*: ACM Press, 2004.

[4] "Unified Modeling Language," http://www.uml.org/, date accessed Nov 2005.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Reprinted ed. Reading, Massachusetts: Addison-Wesley, 1988.

[6] C. Alexander, "The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World," in *IEEE Software*, vol. 16, num. 5, 1999, pp. 71-82.

[7] M. R. Anderberg, *Cluster Analysis for Applications*. New York: Academic Press, 1973.

[8] P. Baldi, P. Frasconi, and P. Smyth, *Modeling the Internet and the Web: Probabilistic Methods and Algorithms*. West Sussex, England: John Wiley and Sons Ltd, 2003.

[9] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," presented at 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992. pp. 59-70.

[10] T. Ball, "The Concept of Dynamic Analysis," presented at 7th European Software Engineering Conference, Toulouse, France, 1999. pp. 216-226.

[11] M. Barnett and W. Schulte, "Spying on Components: A Runtime Verification Technique," presented at Specification and Verification of Component-Based Systems Workshop at OOPSLA 2001, Tampa, FL, USA, 2001. pp. 7-13.

[12] W. Bartussek and D. L. Parnas, "Using assertions about traces to write abstract specifications for software modules," presented at 2nd Conference of the European Cooperation on informatics: information Systems Methodology, 1978. pp. 211-236.

[13] V. R. Basili, S. E. Condon, K. El Emam, R. B. Hendrick, and W. Melo, "Characterizing and modeling the cost of rework in a library of reusable software components," presented at 19th international Conference on Software Engineering, 1997. pp. 282-291.

[14] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM Trans. Comput. Syst.*, vol. 13, pp. 1-31, 1995.

[15] B. Beizer, *Software Testing Techniques*, 2nd ed: Thompson Publishing, 1990.

[16] K. N. Biyani and S. S. Kulkarni, "Building component families to support adaptation," presented at Workshop on Design and evolution of autonomic application software (DEAS), St. Louis, Missouri, 2005. pp. 1-7.

[17] R. Bodik and S. Anik, "Path-sensitive value-flow analysis," presented at 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), San Diego, 1998. pp. 237-251.

[18] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," presented at ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'02), 2002. pp. 2-9.

[19] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "TRIPWIRE: Mediating Software Self-Awareness," presented at SE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04), Edinburgh, Scotland, 2004. pp. 11-14.

[20] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active Learning for Automatic Classification of Software Behavior," presented at International Symposium on Software Testing and Analysis (ISSTA 2004), Boston, Mass., 2004. pp. 195-205.

[21] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, pp. 123–140, 1996.

[22] L. C. Briand, V. R. Basili, and W. M. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis," *IEEE Transactions on Software Engineering*, vol. 18, pp. 931-942, 1992.

[23] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," presented at the 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004. pp. 480-490.

[24] A. Camilleri, *The Shape of Water*, English ed. New York: Penguin Group, 2002.

[25] C. K. Chang, M. J. Christensen, and T. Zhang, "Genetic Algorithms for Project Management," *Annals of Software Engineering*, vol. 11, pp. 107-139, 2001.

[26] B. Coggan, S. D. Crocker, G. Estrin, and D. Hopkins, "Snuper Computer - A computer instrumentation automation," presented at AFIPS Spring Joint Computer Conference, 1967. pp.

[27] D. A. Cohn, L. Atlas, and R. E. Ladner, "Improving Generalization with Active Learning," *Machine Learning*, vol. 15, pp. 201-221, 1994.

[28] J. E. Cook and A. L. Wolf, "Automating process discovery through event-data analysis," presented at Proceedings of the 17th international conference on Software engineering, Seattle, Washington, United States, 1995. pp. 73-82.

[29] E. L. Crow, F. A. Davis, and M. W. Maxfield, *Statistics Manual*. New York: Dover Publications, 1955.

[30] C. Dabrowski and K. Mills, "Understanding self-healing in service-discovery systems," presented at the first workshop on Self-healing systems, 2002. pp. 15-20.

[31] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," presented at the first workshop on Self-healing systems, 2002. pp. 21-26.

[32] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," presented at the 23rd International Conference on Software Engineering, 2001. pp. 339-348.

[33] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering: An International Journal*, vol. 10, pp. 405-435, 2005.

[34] N. Dor, S. Adams, M. Das, and Z. Yang, "Software validation via scalable path-sensitive value flow analysis," presented at 2004 ACM SIGSOFT international Symposium on Software Testing and Analysis, 2004. pp. 12-22.

[35] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. New York: John Wiley and Sons, Inc., 2001.

[36] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," presented at International Conference on Software Engineering, 1999. pp. 213-224.

[37] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The *Daikon* system for dynamic detection of likely invariants," *Science of Computer Programming*, 2006.

[38] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuousvalued attributes for classification learning," presented at International Joint Conferences on Artificial Intelligence, 1993. pp. 1022–1027.

[39] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," presented at Second IEEE International Symposium on Requirements Engineering, York, England, 1995. pp. 140-147.

[40] S. D. Fleming, B. H. C. Cheng, R. E. K. Stirewalt, and P. K. McKinley, "An approach to implementing dynamic adaptation in C++," presented at Workshop on Design and evolution of autonomic application software, St. Louis, Missouri, 2005. pp. 1-7.

[41] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," presented at 1996 IEEE Symposium on Security and Privacy, 1996. pp. 120-128.

[42] D. Garlan and B. Schmerl, "Model-based adaptation for selfhealing systems," presented at the first workshop on Self-healing systems, 2002. pp. 27-32.

[43] D. Garlan, M. Litoiu, H. A. Muller, J. Mylopoulos, D. B. Smith, and K. Wong, *DEAS 2005: workshop on the design and evolution of autonomic application software*: ACM Press, 2005.

[44] "The GCC Home Page," http://www.gnu.org/software/gcc/, date accessed June 2006.

[45] R. Griffith and G. Kaiser, "Manipulating managed execution runtimes to support self-healing systems," presented at Workshop on Design and evolution of autonomic application software (DEAS), St. Louis, Missouri, 2005. pp. 1-7.

[46] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. Votta, "Proactive system maintenance using software telemetry," 2003, pp. 24-26.

[47] L. K. Hansen and P. Salamon, "Neural Network Ensembles," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, pp. 993-1001, 1990.

[48] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying Classification Techniques to Remotely-Collected Program Execution Data," presented at European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), 2005. pp.

[49] M. Harder, J. Mellen, and M. D. Ernst, "Improving Test Suites via Operational Abstraction," presented at the 25th International Conference on Software Engineering, 2003. pp. 60-71.

[50] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," presented at ACM SIGSOFT international Symposium on Software Testing and Analysis, 1998. pp. 11-20.

[51] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of the Relationship Between Fault-Revealing Test Behavior and Differences in Program Spectra," *Journal of Software Testing, Verifications, and Reliability*, vol. 10, 2000.

[52] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," presented at 16th International Conference on Software Engineering, 1994. pp. 191–200.

[53] "Autonomic computing," http://www.research.ibm.com/autonomic/, date accessed June 2005.

[54] J. A. Jones, A. Orso, and M. J. Harrold, "Visualization of test information to assist fault localization," presented at 24th International Conference on Software Engineering, 2002. pp. 467-477.

[55] D. E. Knuth, *The Art of Computer Programming*, vol. 1, 2nd ed: Addison-Wesley, 1973.

[56] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *Bit*, vol. 13, pp. 313-322, 1973.

[57] J. A. Kowal, *Behavior Models: Specifying User's Expectations*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[58] V. G. Kulkarni, *Modeling, Analysis, Design, and Control of Stochastic Systems*. New York: Springer-Verlag, 1999.

[59] W. Landi and B. G. Ryder, "Pointer-Induced Aliasing: A Problem Classification," presented at Eighteenth Annual ACM Symposium on Principles of Programming Languages, 1991. pp. 93-103.

[60] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. SE-9, pp. 347-354, 1983.

[61] D. Leon, A. Podgurski, and L. J. White, "Multivariate visualization in observation-based testing," presented at the 22nd international conference on Software engineering, 2000. pp. 116-125.

[62] D. Liang and K. Xu, "Program Execution Monitoring with Scenario Implementation Models," University of Minnesota, Minneapolis, MN, Technical Report TR 04-034, September 2004.

[63] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," presented at Programming Language Design and Implementation (PLDI), 2005. pp.

[64] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions in Programming Languages and Systems*, vol. 16, pp. 1811-1841, 1994.

[65] N. Littlestone and M. K. Warmuth, "The Weighted Majority Algorithm," 1989, pp. 256-261.

[66] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," presented at Conference on Software Maintenance, 1993. pp. 282-291.

[67] T. J. McCabe, "A complexity measure.," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308-320, 1976.

[68] G. J. Meyers, *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

[69] T. M. Mitchell, *Machine Learning*. Boston: McGraw-Hill, 1997.

[70] J. C. Munson and S. Elbaum, "Software reliability as a function of user execution patterns," presented at the Thirty-second Annual Hawaii International Conference on System Sciences, 1999. pp. 8004-8014.

[71] J. W. Murdock and A. K. Goel, "Meta-case-Based Reasoning: Using Functional Models to Adapt Case-Based Agents," presented at the 4th international Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development, 2001. pp. 407-421.

[72] J. Musa, *Software Reliability Engineering*. New York: McGraw-Hill, 1999.

[73] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," 2002, pp. 65-69.

[74] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," presented at the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), 2003. pp. 128-137.

[75] D. L. Parnas, "Precise description and specification of software," presented at Second international Conference on Mathematics of Dependable Systems II, Univ. of York, England. pp. 1-14.

[76] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," presented at 21st International Conference on Software Engineering (ICSE'99), 1999. pp. 277-284.

[77] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and its Implications for Software Testing," *IEEE Transactions on Software Engineering*, vol. 16, pp. 965-979, 1990.

[78] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang, "Estimation of software reliability by stratified sampling," *ACM Transactions on Software Engineering and Methodology*, vol. 8, pp. 263-283, 1999.

[79] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," presented at ICSE '03: the 25th International Conference on Software Engineering, 2003. pp. 465-475.

[80] R. S. Pressman, *Software Engineering: A Practioner's Approach*. New York: McGraw-Hill, 1992.

[81] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley, 1999.

[82] L. Rabiner and B. Juang, *Fundamentals of Speech Recognition*. New Jersey: Prentice Hall, 1993.

[83] G. Ramalingam, "Data flow frequency analysis," presented at Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, Philadelphia, 1996. pp. 267-277.

[84] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for program test data selection," presented at 6th International Conference on Software Engineering, 1982. pp. 272-278.

[85] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM Software Engineering Notes*, vol. 22, pp. 432-439, 1997.

[86] K. H. Rosen, *Discrete Mathematics and its Applications*, Third ed. New York: McGraw-Hill, Inc., 1995.

[87] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, pp. 401-419, 1998.

[88] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, New Jersey: Pearson Education, Inc., 2003.

[89] R. E. Schapire, "The Strength of Weak Learnability," *Machine Learning*, vol. 5, pp. 197-227, 1990.

[90] R. E. Schapire, "Theoretical Views of Boosting and Applications," vol. 1720: Springer, 1999, pp. 13-25.

[91] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney, "Model-carrying code: a practical approach for safe execution of untrusted applications," presented at 19th ACM symposium on Operating systems principles, 2003. pp. 15-28.

[92] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, pp. 209-254, 2001.

[93] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal*, vol. May, 1974.

[94] H. M. Taylor and S. Karlin, *An Introduction to Stochastic Modeling*, 3rd ed. San Diego, CA.: Academic Press, 1998.

[95] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," presented at International Conference on Software Maintenance, 1998. pp. 44-53.

[96] D. W. Wall, "Predicting program behavior using real or estimated profiles," presented at '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, 1991. pp. 59-70.

[97] G. M. Weinberg, *PL/I Programming Primer*. New York: McGraw-Hill, 1966.

[98] E. J. Weyuker, "The Applicability of Program Schema Results to Programs," *International Journal of Computer and Information Sciences*, vol. 8, pp. 387-403, 1979.

[99] J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 93-106, 1996.

[100] T. Xie and D. Notkin, "Checking inside the black box: Regression testing based on value spectra differences," presented at the International Conference on Software Maintenance, Chicago, 2004. pp. 28-37.

[101] D. Zhang and J. J. P. Tsai, "Machine Learning and Software Engineering," presented at 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02), 2002. pp. 22-29.